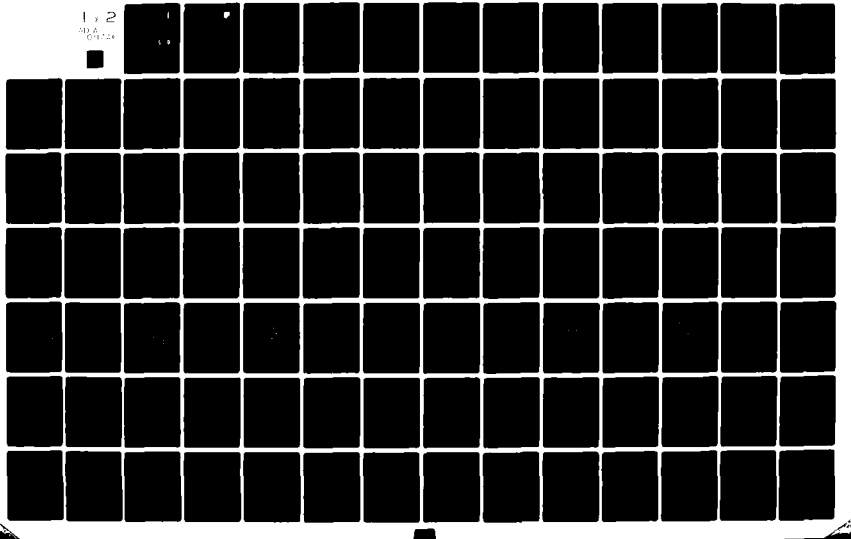
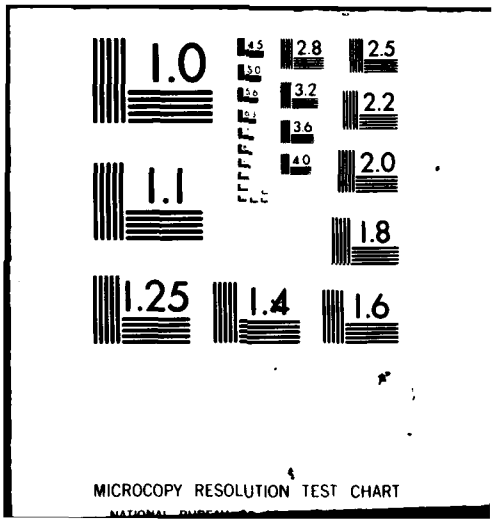
AD-A109 746 INTERMETRICS INC CAMBRIDGE MA F/G 9/2
ADA INTEGRATED ENVIRONMENT I. DESIGN RATIONALE.(U)
DEC 81 F30602-80-C-0291
UNCLASSIFIED RADC-TR-81-357 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

PHOTOGRAPH THIS SHEET

AD A109746

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Intermetrics, Inc.
Cambridge, MA

ADA Integrated Environment I
Design Rationale
Dec. 81

DOCUMENT IDENTIFICATION

Contract F30602-80-C-0291 RADC-TR-81-357

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
S JAN 19 1982 D
D

DATE ACCESSIONED

82 01 12 001

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-357
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I DESIGN RATIONALE

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-357 has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-357	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT I DESIGN RATIONALE		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0291
9. PERFORMING ORGANIZATION NAME AND ADDRESS Intermetrics, Inc. 733 Concord Avenue Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F 55811908
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981 13. NUMBER OF PAGES 141
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This report describes the rationale of the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an APSE is

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this report include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

PREFACE

The construction of an Ada Integrated Environment is the logical next step in the progression of efforts to solve "the software problem", the most recent step being the development of the Ada language itself. This design for an Ada Integrated Environment is the result of a team effort by Intermetrics, Incorporated and Massachusetts Computer Associates, Inc. (COMPASS), with Dr. Fred Martin as Project Manager.

Intermetrics team members have included Arra Avakian (Technical Director), Ben Brosgol, Morris Kranc, Rich Peterson, Tucker Taft, and Michael Tighe. Additional significant contributions were made by Mark Davis, Karen Huff, Ron Kole, David Levine, Stavros Macrakis, and Mike Ryer.

COMPASS team members have included David Loveman (COMPASS coordinator), Tolly Holt, Charley Muntz, Mat Myszewski, and Steve Schuman. Additional significant contributions were made by Paul Cashman, Mark Marcus, David Presberg, Kirk Sattley, and Stu Schaffner.

Outstanding logistical support, coordinated by Toby Boyd, was provided by Ruth Barton, Robin Camardo, Valerie Censabella, Tim Fiorino, LeAnne Grillo, John Heymann, Elizabeth LeProux, Deirdre Munro and Cindy Paige, of Intermetrics; and Jim Botto, Kate Russell, and Sue Woodyard, of COMPASS.

The design team wishes to specifically acknowledge Jean Ichbiah for the design of Ada, John Buxton for the development of the Stoneman document, members of the Intermetrics DARPA Ada compiler project for valuable insights, and the researchers and developers too numerous to mention in the fields of programming environments, tool design, and compiler construction whose ideas we have utilized.

This design effort was sponsored by the United States Air Force, Rome Air Development Center. Don Roberts served as the contract monitor.

TABLE OF CONTENTS

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Chapter 1

EXECUTIVE SUMMARY

The stated objective of the Intermetrics/COMPASS design is to specify a product to solve real software-development problems in a practical, efficient and user-friendly manner. Our priorities in approaching many of the difficult tradeoff decisions in satisfying the Statement of Work [S.O.W.] have been: to serve a wide range of potential users (including managers, maintainers and novice to expert programmers, in large and small projects); to ensure the practicality and implementability of the design; and to tailor support to the preparation, compilation, execution and maintenance of Ada programs.

A definite theme running through this effort is the application of a few general principles, ideas or techniques to the realization of the first instance of a minimal Ada Programming Support Environment, viz the MAPSE. As a result, a high degree of integration and integrated communications are evidenced across data, control and development concepts. STONEMAN [STONEMAN] and the S.O.W. suggest a layered implementation of virtual machines where the Kernel APSE (KAPSE) isolates all host machine dependencies from the user and then presents a universal (Ada) interface to software tools and users alike. The KAPSE becomes the medium of exchange, through the database and other facilities, for communications and control between users and/or tools. This is exactly the approach taken in this design. All host dependencies are, in fact, isolated within a few KAPSE routines. The KAPSE is 'keeper of the database' and controls and monitors all accesses. All other Ada programs, whether MAPSE tools or user-written Ada programs, in a sense, have the same status. Each looks to the KAPSE for run-time support, for data, and for communications with other tools.

The principal user interface is the Command Processor (CP), which provides a natural <verb><object> interactive MAPSE Command Language (MCL), through which all MAPSE tools are available. The CP responds to both user keyboard (or batch) inputs as well as stored scripts. Since all MCL commands can be effected using Ada statements as well, the MAPSE can be equally controlled from any executing Ada program as from the CP. The CP standardizes conventions between tools: most tools will accept input from a standard file and direct output to a standard file. This allows the CP to chain the operation of one tool to another without the user having to designate intermediate files. This chaining or piping

is a great convenience in preparing user scripts or complex interactive command sequences. Of course, the user can always redirect input or output from/to specific files if he so chooses.

From the CP, the user may select a powerful set of MAPSE tools: the text editor for preparing general text or Ada source (the editor has an Ada mode which recognizes Ada lexical elements); the Ada compiler for compiling single or multiple compilation units into the program library; the linker for collecting consistent sets of computation units for execution; and the debugger which controls the execution of Ada programs and provides the necessary diagnostic tools for development.

KAPSE/DATABASE

The key elements of the MAPSE are the KAPSE/Database and the Ada compiler. The singular contribution of the database design is the notion of "distinguishing attributes". Here the user can arbitrarily set up a multi-dimensional name space where each name is defined by a set of distinguishing attributes. Thus, for example, a project may choose to establish a 5-space indicating (aircraft, subsystem, module, test-status, type). Values for these attributes can define names or, "locations", in the database for source, documentation, test scripts, etc. (These could be within the Ada program library, for the library itself must be part of the database.)

In order to see the power of this concept, consider the following database entities from a hypothetical B-52 upgrade development:

```
(B_52, power, pl06_A. level_0, source)
(B_52, navigation, controller, level_1, test_script)
(B_52, weapon, search, config_A, source)
(B_52, navigation, GPS_signal, level_0, source)
```

Note that from the B_52 root this can be viewed (vertically) as an hierarchical configuration, or tree. However, the KAPSE/Database facilities also permit arbitrary (horizontal) partitions of the same data; i.e., all instances of source can be accessed or manipulated or, for example, all instances of level_1 scripts, etc.

Arbitrary and flexible naming is only one aspect of this design. This is coupled with access controls to satisfy all the database requirements in the SOW. The KAPSE implements the concept of a "window" on the database where the window carries with it a role for its user. The role (e.g., editor, manager, tester, etc.) is then translated into particular operational abilities to deal with the data being accessed. Thus, in the examples above, the B_52 project manager may allocate separate windows to power, navigation and weapons groups for software development, e.g.,

(B_52, power, *, *, *)

Within these groups more restrictive windows may be further differentiated with constrained capacities, e.g., read-only, read/write, edit, test, etc. A small test sub-group might have access to only

(B_52, power, *, *, test_script)

or an IVV contractor might be able to address all test scripts for config_A releases; i.e.,

(B_52, *, *, config_A, test_script)

Of course, the combinations are endless but these simple concepts satisfy all of the SOW requirements on the database: management and control. The system allows a straightforward and conventional organization while having within it the potential for developing the necessary structures and controls to support a variety of management disciplines. Responsibilities can be highly centralized or widely distributed.

One point to be added is that each database object has appended to it a history file of how it got to this point, and a category which allows system interpretation of the data and how it may be used.

It is anticipated that objects in the database such as Ada program libraries, may become quite large and contain data structures of arbitrary complexity. Program access to these database objects is supported, therefore, by a virtual memory management (VMM) system which permits logical structuring of the data for machine-independent access and manipulation. By use of a program the VMM can store, create, modify, and retrieve extremely large data structures which exceed the memory space of the computer. The designer and programmer are free to concentrate on the logical data structure; the VMM system takes care of the physical structure. VMM will prove invaluable in aiding the rehosting of the MAPSE to small machines.

ADA COMPILATION

The other key element in the system is the compiler and the methods by which compilation is accomplished. Since the MAPSE itself (i.e., most of the KAPSE and all of the tools) is written in Ada, the first requirement is that the compiler issue efficient code and that the run time system (RTS) be especially efficient and support full Ada generality. The entire system depends on and constantly uses the RTS; its efficiency is more important to practical system utility than esoteric optimization techniques.

The approach taken by Intermetrics/COMPASS in designing the compiler is conservative and one with which success has already been demonstrated. Based on PQC/TCOL ideas [Leverett 80], the compiler takes the form of a series of independent phases which add to and/or modify information in an intermediate language (IL) definition at each interface. The design embraces the new Diana definition [Goos 81] as the intermediate language. The structure of Diana then remains constant from semantic analysis right through code generation. Maximum flexibility is maintained by delaying optimization enhancements until usage patterns can be established and analyzed. The use of Diana instead of TCOL as an intermediate language is but a variation on the already established Intermetrics compiler building techniques. The front end and run time system of the MAPSE/Ada compiler is based on the DARPA Ada compiler being built by Intermetrics and now nearing release. The middle phases are a reflection of the Intermetrics one-year participation in the PQCC project at CMU. The back end compilation technique derive directly from the Intermetrics Standard Compiler (ISC) approach, which has successfully been applied to the development and delivery of Pascal, FORTRAN, and JOVIAL compilers. In summary, our rationale for selecting this design is simple - we know it will work; the technical risk is minimal.

Ada supports separate compilations. A complete embedded computer system (ECS) software product might involve hundreds of interdependent compilation units and thousands of compilations and re-compilations. It is imperative that these be conducted in an efficient manner. The potential problem can be illustrated by considering a series of compilation units, A through Z, with interdependencies, e.g.,

```
A with B
B with C
C with D
.
.
.
Y with Z
```

In this case A through Y depend upon Z, A through X depend upon Y, With this example as a model, the recompile strategy of the design will significantly reduce the potential for extraneous compilations by delaying the necessity for creating a consistent program library until the library is actually used, i.e., linked. This allows repeated intermediate compilation steps without requiring immediate recompilations of all potentially affected units. (See Program Integration B-5 Specifications, Section 6 for a more elaborate discussion of this problem and its solution.)

EMBEDDED COMPUTER SYSTEM DEVELOPMENT

The main object of an APSE is the development of embedded computer software. The database, compiler and management facilities are necessary supports but the resulting software must execute and be verified while executing. The philosophy here is consistent with STONEMAN in that ECS software should be developed and verified to the greatest extent possible on an APSE host before commitment to a target machine. With target machine diagnostic facilities usually a minimum both in capacity and availability, this makes great sense for large multi-programmer efforts. The MAPSE design presented here integrates debugging and simulation such that it supports the spectrum of testing from interactive unit debugging to large scale digital simulation, either on-line or batch. Debugging is accomplished through a set of MAPSE capabilities commanded by syntax extensions to MCL. The user therefore can employ the full capabilities of MCL within the debugging environment. As a result, the full power of the MAPSE (i.e., all tools) can be brought to bear on a debugging problem, in interactive or batch mode.

The approach is to take advantage of the compiler DEBUG option which instruments the code and provides potential breakpoints ('hooks') at every Ada statement. The penalty is small for interactive debugging on the host but the benefit is large when the code is integrated and verified against environment simulators, controlled by batch scripts or interactive commands. The MAPSE design further lays the foundation for 'functional simulation' as a future APSE tool, a technique whereby Ada statements executing on the host are weighted with target machine elapsed times. Time is then accumulated as a software pseudo-time quantity and Ada tasking and environmental simulators are made consistent with this time. The result is an effective closed-loop digital simulator for verification and validation of ECS software on the host. (This basic technique was pioneered by Intermetrics and has been in successful use for more than 8 years in the development of Shuttle Software using HAL/S). Final verification of ECS software takes place on the target machine itself which can be supported by the host debugger and database. Although the intention is for target debugging to have the same functionality as host debugging, specific considerations are beyond the scope of the current MAPSE design effort.

MAPSE ENVIRONMENT

Our final point to be made in this overview of MAPSE design highlights is the utilization of resources on the 370/VM and Perkin Elmer 8/32 host computers. Users on each machine may operate in relative isolation from each other. On the 370/VM each user logging into the MAPSE will receive a separate 370 virtual machine (VM). All MAPSE tools are available on every VM and are supported by instances of the Ada run time system. In a sense, each tool (or Ada program) has its own Ada virtual machine. The

result is that independent users can develop, i.e., compile, execute, debug, etc., their own Ada programs in isolation. In this model, the database and database operations also reside within a VM and data accesses are accomplished by the 370/VM system Virtual Machine Communication Facility (VMCF), an efficient high speed memory-to-memory pathway. Note that this MAPSE allocation leaves the 370/VM machine free to support non-APSE users on other VM's. In this way, corporate 370's need not be dedicated to MAPSE use. Scheduling and utilizations of all VM's are handled automatically by the 370/VM operating system, CP-VM370. Where interface between MAPSE and non-MAPSE is required, the KAPSE will utilize an import/export function provided for that purpose.

Mapping onto the Perkin Elmer machine is done analogously. Each user gets an independent OS/32 time-sliced by OS/32. From these tasks MAPSE tools can be activated also as independent OS/32 tasks. A relationship among tools being used by a particular user is maintained by the KAPSE. Database access is handled by the OS/32 Intertask Communications Facility. In this case, however, the KAPSE will rely on OS/32 file handling and disk I/O routines for mass storage. As with the 370/VM, the MAPSE on this machine is a 'good neighbor'. That is, non-MAPSE users can coexist, simply by being allocated separate tasks.

The Rationale document (IR-684) explores most of the MAPSE design decisions in considerable detail. The overall MAPSE architecture is presented in the System Specification (IR-676). The detailed specifications are found in the following Computer Program Development Specifications:

-Ada Compiler (IR-677)
-KAPSE/Database (IR-678)
-MAPSE Command Processor (IR-679)
-MAPSE Generation and Support (IR-680)
-Program Integration Facilities (IR-681)
-MAPSE Debugging Facilities (IR-682)
-MAPSE Text Editor (IR-683)

Within the time constraints of six calendar months, the Intermetrics/COMPASS team has put forth an honest effort toward designing for implementation, the best Ada programming environment possible, complying with the requirements of the S.O.W. and the guidelines of Stoneman. We believe we have succeeded and respectfully offer this design for your evaluation.

Dr. Fred H. Martin
Project Manager
March 15, 1981
Cambridge, MA

Chapter 2

INTRODUCTION

This Rationale document, together with the companion System Specification and Computer Program Development Specifications present the design of a minimal Ada Programming Support Environment - MAPSE - as called for in the RADC Statement of Work PR No. B-0-3233. The design is also consistent with the general guidelines presented in the DoD sponsored STONEMAN [STONEMAN]. The MAPSE is the basis for the implementation of a more complete APSE and, eventually, a comprehensive Ada Integrated Environment (AIE).

BACKGROUND

The situation that has given rise to the need for an Ada Integrated Environment is of major significance to the DoD. We see that situation in the following broad terms.

The revolution in computer hardware has made it possible to include computer elements in virtually every military technical configuration. However, this potential has not been realized. The problem has been in achieving achieving the desired cost effective and timely production of reliable and maintainable large-scale software. STONEMAN comments on this, as follows:

> "Typically hardware costs now account for only some 15% of project costs, with 70% to 90% of software costs arising in the long term life cycle maintenance and support phase of the system".
>
> [STONEMAN, 2.A.1]

Decades of effort and vast sums have gradually brought home the lesson that no single (let alone simple) prescription will substantially change the situation - not "automatic programming", not a programming language alone, nor a software engineering methodology, etc. In sponsoring the MAPSE development as a first move towards an AIE, the Air Force has recognized the "holistic" and intrinsically complex character of the needed achievement.

The focus on a computer-supported integrated environment - for all life cycle phases of software systems - as a target of design and implementation is a bold but technically sound initiative. Although not numerous, there already exist environment projects now sufficiently mature (e.g., PDS [Cheatham 79], Mesa

[Lauer 79] and [Horsley 79], Gandalf [Notkin 79], CADES [McGuffin 79], NSW [Millstein 77]) to demonstrate that (a) designed software environments can indeed make significant improvements in software production efficiency and quality; (b) the technological basis for achieving the basic goals of the RADC Statement of Work (S.O.W.) and the STONEMAN already exists.

EMBEDDED COMPUTER SYSTEMS (ECS) ACTIVITIES

The development and maintenance of ECS software is a significant technical - industrial activity, comparable in scope to other major military technologies. A single project over its lifetime may well involve hundreds, if not thousands of technically and managerially concerned participants, over periods that may be measured in decades.

As with all important military technologies, the needs for ECS software reliability and adaptability to changing field requirements can hardly be overstated. The reliability demand implies well-developed and controlled quality assurance procedures as part of the development process. Reliability and adaptability together translate into enforced modularization disciplines which promote maintenance and adaptation. Indeed, this is an important component in the rationale for the Ada language in the first place.

The demands for reliability and adaptability - never easy to meet - are made more difficult by the fact that typically, ECS software (a) must meet critical real time requirements; (b) is event driven, and therefore entails the coordination of every asynchronous process; (c) will have to execute on computers with little excess capacity - and, for some applications, on multiple processors, more or less distributed. Every one of these conditions adds greatly to the difficulty of building robust, maintainable ECS software.

Yet another difficulty stems from the fact that ECS software must fit into an unstable hardware environment. The hardware too will tend to be at the cutting edge of technology, and therefore subject to corrections and improvement.

EXPECTED AIE LEVERAGE

What are the main sources of leverage in an integrated environment - such as the required MAPSE? The first is controlled, long-term-adaptable, and easy-to-understand integrated communication between people and people, people and tools, and tools and tools. Next are powerful tools of high quality which are integrated with each other with respect to usage, data, and controls. Finally, there is stability in the face of host and target perturbations, that is, replacement of host or target by another machine (the portability requirements).

Of these properties, integrated communication is the hardest to achieve in the present state of the art. However, it is the property upon which growth from the MAPSE to a full-blown AIE critically depends.

RELEVANCE TO THE SOFTWARE BOTTLENECK

There is every reason to believe that the above-named properties, and especially the communications property, correctly address the software bottleneck. Consider the following points:

a. Large scale software development involves not only many applications programmers and their managers, but others as well. During development, ongoing communication must be maintained with system specifiers and designers, hardware engineers, etc. [STONEMAN, 2.A.11.a]. For debugging and later system maintenance, audit trails of these communications and their effect on the software need to be logged. Doing this without computer-aided communications would be extremely difficult.

b. Embedded Computer Systems (ECS) software maintenance is typically long term, and distributed over many systems in the field. Here extensive communications requirements between the maintenance organizations and the user community arise. Without computer environment support, these communications would be very costly and difficult to manage [Cashman 80]. In addition, because of the long life of military ECS systems, one can expect a substantial turn-over in personnel responsible for the development and maintenance of a single system. This is a powerful motive for record keeping and controlled communications as part of system evolution.

   The essence of the configuration management problem, critical in every life cycle phase of a system, is this communication and coordination among the many role players concerned with the various parts of a configuration [STONEMAN, 2.A.15].

c. The military requires many formal steps of documentation, review, sign-off, etc. All of these call for structured and controlled communications [STONEMAN, 2.A.3].

d. Clearly, powerful tools are required in a large, multi-faceted activity such as software production. This is especially true of the real-time, high-reliability, applications that characterize ECS software [STONEMAN, 2.A.11.b]. Sophisticated debugging, simulation, compiling tools, etc. are indispensable.

e. Stability in the face of host perturbations is of great importance to environments for ECS software development. The development of ECS software and its subsequent support extend over long periods of time, and can involve many people at multiple locations. Host changes are inevitable.

Finally, the reader is reminded of the assertion implicit in STONEMAN (in 2.A.1) that 60% to 75% of total ECS system costs are unrelated to its hardware or original development, but to its long-term software maintenance support. Bringing the maintenance and support aspects of ECS software under control depends, most of all, upon communications and record keeping controls throughout the system's life - from its original specification forward. This is a service which nothing but an integrated environment can render. While the MAPSE is not supposed to tackle this problem directly, it must provide the foundation for its solution.

ADA ENVIRONMENT TERMINOLOGY

STONEMAN introduced a number of terms for the discussion of Ada environments:

APSE a "full" Ada programming support environment.

Although STONEMAN identifies the APSE as support for "the development and maintenance of Ada applications software throughout its life cycle" [STONEMAN, 1.B], the term APSE emphasizes the programming activity above such other required activities as specifying, system modelling, documenting, functional testing, etc. The term Ada Integrated Environment (AIE) which appears as the title of the S.O.W. corrects this bias. It is apparent from the text of STONEMAN that the full APSE is indeed supposed to encompass all of these activities in an integrated fashion.

MAPSE This is a minimal APSE specified by STONEMAN and the S.O.W. to include the most basic tools of current day programming - linker/loader, command interpreter, editor, compiler, debugger - and some tools for configuration management, an activity indispensable to system development and maintenance.

Although the MAPSE is not specified to contain supporting tools for all life cycle phases of ECS software, it must nevertheless exhibit the basic capability of integrated environments: to be a medium for coordinating diverse activities - a hard enough problem even when only MAPSE supported activities are considered.

According to STONEMAN, an APSE grows out of a MAPSE by

tool augmentation, but (ideally) by no other structural changes. As suggested by the S.O.W., we shall view MAPSE's as well as APSE's as instances of AIE's.

KAPSE the kernel of an APSE (or MAPSE) - analogous to an operating system.

The kernel is the framework within which users, their tools and their programs cooperate. Its content is further detailed below.

PERSPECTIVE ON MAPSE REQUIREMENTS

Most of the MAPSE requirements in the S.O.W. and Stoneman stem from the character of the activities which the MAPSE - and later APSE will have to support. Because the MAPSE must be capable of developing into an APSE, all APSE activities must, to some extent, be considered from the beginning. These activities include those performed by a variety of people, including: analysts, programmers, managers, documenters, etc.

The requirements naturally fall into four categories: integration, tools, rehostability and retargetability, testing and maintenance.

Integration

The first of four integration requirements is support for Ada. This requires that the environment provide tools for the preparation, coding and execution of Ada programs. In addition, the MAPSE should be programmed in Ada and further, should be utilized for the ongoing maintenance and support of itself.

The second major integration requirement is that the KAPSE database be specified as "the central feature of the MAPSE".

The third major integration requirement is the need for a uniform interface between any intercommunicating entities in the system. This is to be accomplished by means of the KAPSE; that is to say, the KAPSE is to supply a "virtual interface or standard interfaces" which guarantees wide ranging intercommunicability.

As much as possible also, the S.O.W. demands that intermediate data developed by one tool but possibly useful to another be recognized, specified, and standardized so as to maximize tool compatibility.

The fourth major integration requirement is the need for integrated project and configuration management. This can only be effected if there are APSE supported methods for defining project structure in a strong sense: defining the protocols to be followed in respect to computer operations of various project-

related functions - definitions which the APSE can use for tracing and/or controlling the development of computer stored objects and object collections.

We therefore state: It is a requirement on the KAPSE design that it support the evolution of project structure definitions - definitions which enhance the ability to trace and/or control the connections of cause to effect in the processes of ECS software development and maintenance.

Tools

The fundamental tool requirements deal with tool construction and management rather than detailed tool operation. These requirements include implementation in Ada, conformance to standard coding and management styles, and uniform and consistent format and communication protocols. The one tool upon which the entire MAPSE rests is the compiler.

The Ada compiling facility is as fundamental to the MAPSE - and later APSE - as the MAPSE/database. Since compiling is required to make any Ada program executable, and since the MAPSE itself largely consists of Ada programs, the performance characteristics and capabilities of the compiler influence all performance characteristics and capabilities of the MAPSE taken as a whole.

Rehosting and Retargeting

Rehosting and retargeting must cover any change in the hardware/software host (target) which can affect the KAPSE/MAPSE software. Such changes may be relatively minor perturbations - such as a new field release of the underlying operating system (if present), or a change in the mass storage medium visible to the MAPSE. On the other hand, there might be a total replacement of host or target.

The implications of this requirement are far-reaching. First, every Ada program in executable form must include the Ada run-time package. In addition, the KAPSE, which must use host resources with acceptable efficiency, will contain host-dependent procedures. The design of the KAPSE must therefore include a partition of the KAPSE into modules so that rehosting will either involve:

- changes in the run-time package alone, or
- KAPSE recompilation, or
- replacement of some number of KAPSE host-dependent procedures

depending on the severity of the change in the host.

The KAPSE will be the most host-sensitive portion of the MAPSE. The MAPSE tools - and later APSE tools - though written in Ada, and protected by the KAPSE, will nevertheless require careful management in order to exhibit the desired rehostability characteristics.

Testing and Maintenance

The problem of testing and maintaining software can be briefly stated as: to ensure that the delivered configuration correctly and consistently provides the required system. The delivered configuration includes:

- requirements,
- designs,
- source modules and command procedures,
- executables, and
- documents.

Configuration items will be developed, tested, and accepted, but will change over time and require frequent retest. The MAPSE requirement is: to provide effective testing capability in a manner which minimizes life-cycle expense. Testing support is needed for each stage of the life cycle:

Life Cycle Stage	Test Support
Requirements	Analysis Tools
Design	Requirements tracing tools
Coding	Static analysis tools
Execution	Dynamic analysis tools
Acceptance	Test assessment
Maintenance	Re-test drivers and evaluators
Evolution	All of the above

Many of these needs will be served by analytical tools: requirements analysis and tracing, static code analysis, etc. These are not to be part of the MAPSE, but are examples of tools to be found in the more mature APSE.

The following is a list of test scenarios - less those of advanced APSE tools - which the MAPSE must support:

- Configure a test environment including supporting stubs, drivers, etc.

- Test a module in its test environment.

- Apply test scripts to a (relatively complete) program, Save results for comparing subsequent re-tests,

- Retest a program. Automatically compare with previous results.

MAPSE ARCHITECTURE PREVIEW

The architecture of the proposed MAPSE is an elaboration of the concept presented in STONEMAN, and required in the Statement of Work, as illustrated in Figure 2-1.

Functionally, the KAPSE provides the framework for all communication. Its design is therefore the foundation for the integrated communication property referred to above. As the S.O.W. proposes, the database which is the central feature of the KAPSE serves as the primary medium through which MAPSE components communicate. In our design, it is the medium through which all communication flows - programs and users, in all combinations. It is therefore apparent that the KAPSE design is critical to various fundamental capabilities of the resulting AIE; in particular (a) to meet the needs of managers; (b) to develop tool sets as intercommunicating tool modules; (c) to augment the capabilities of individual users.

The KAPSE also plays the critical role of properly containing the efforts of host dependence, so that the MAPSE and its later extensions will enjoy the required stability in the face of host system modification, or even replacement (the portability requirement).

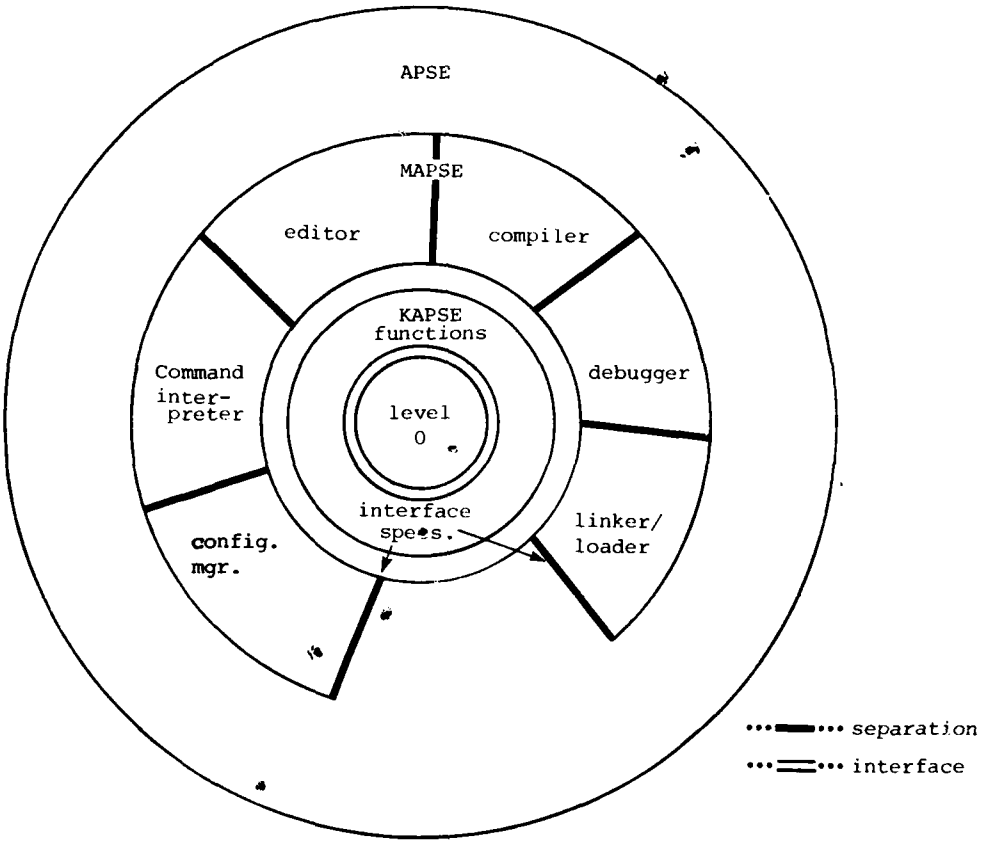
In a sense, the KAPSE can be thought of as a virtual operating system, with special emphasis on intercommunication among users and their programs. Traditionally, operating systems were mainly focused on insulating users from each other. While mutual non-interference is an important aspect of the KAPSE, appropriately controlled mutual contact must also be an objective. This is the point of view from which the KAPSE design should be examined and evaluated.

Discussion of the functional aspects of MAPSE tools is deferred to later chapters.

We shall now re-present the MAPSE architecture to take explicit account of:

a. user interfaces

b. run-time support of executing Ada programs



Level 0

- host hardware/software

Level KAPSE

- host interface
- database facility
- program execution facility
- I/O facility
- User operating (including log-on/off) facility
- interfaces
  - user to tools and other programs
  - user to user
  - program to program (including tool to tool)

Figure 2-1 (Equivalent to Stoneman 1.F)

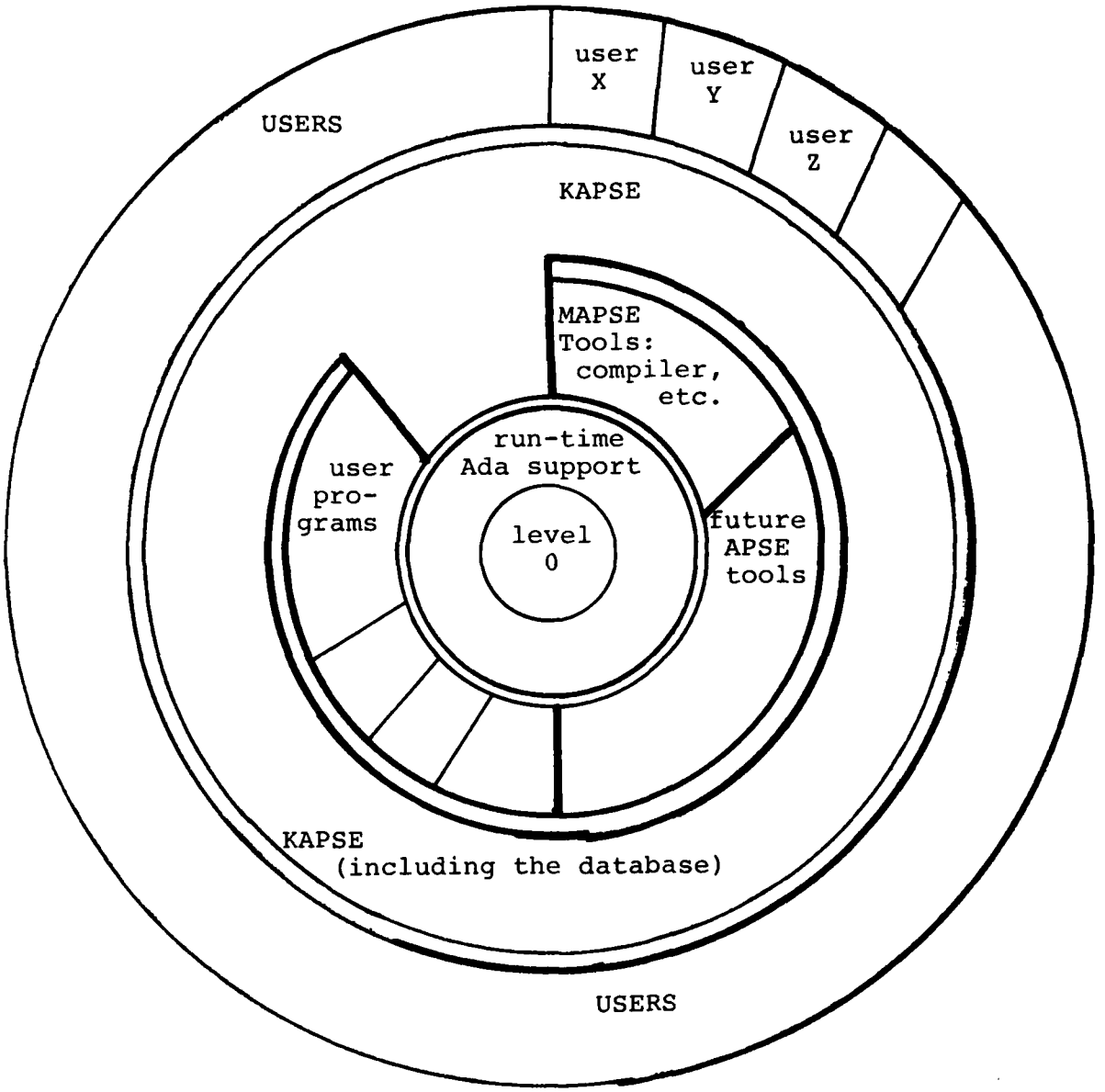


Figure 2-2

The features especially to be noted in Figure 2-2 are:

a. that the KAPSE and its database are like a telephone central which interconnects all tools, user programs and users to one another,

b. that the KAPSE - as an Ada program - uses the run-time support facility, as does every other executing Ada program, and

c. that the APSE grows out of the MAPSE by nothing more than the addition of tools.

Figure 2-2 will be the subject of further refinement and discussion in Chapter 3.

GENERAL ORGANIZATION OF THE RATIONALE

After this Introduction there follow eight chapters which present the major features of the design and associated rationale, feature-by-feature. Of these the first four chapters deal with the MAPSE operational framework: MAPSE overall architecture, KAPSE, facilities for program integration and the command processor. Four more chapters deal with the remaining MAPSE tools, including the compiler.

Throughout the rationale, use will be made of example programming, management or maintenance scenarios with explanations as to how our design supports their realization.

Over and above achieving the AIE leverages described above, we focused on producing a design which gives:

- Usability - a helpful, easy-to-understand, uniform, nuisance-free interface to the user.

- Open endedness - solid foundation for AIE evolution - in the direction of more varied and more powerful tools, covering more of the life-cycle phases of the system and addressing more management concerns.

- Technical reliability - using only proven software technology, well constructed from the testing and maintenance point of view.

We believe that all S.O.W. requirements have, as their rationale, the properties we have discussed. We hope to demonstrate that our design meets the S.O.W. requirements in particular, as well as the larger issues to which they are addressed.

Chapter 3

MAPSE ARCHITECTURE

STATEMENT OF THE PROBLEM

The MAPSE objective can be summarized briefly as "produce a MAPSE which is both usable for Ada program development and support and open ended so that it can be extended to an APSE and, eventually, to a full AIE."

Design Overview

Our design for a MAPSE contains a number of distinctive features which, in an integrated manner, contribute to the potentially conflicting objectives of immediate usability and open endedness. Although these features are discussed in this and subsequent chapters, it is worth highlighting them here:

- All MAPSE tools are written in Ada in a style which avoids Ada machine-dependent facilities. Thus any MAPSE tool may be maintained in a MAPSE, and transported simply by recompilation.

- The KAPSE is written in Ada. Most KAPSE modules are written in the same style as MAPSE tools, facilitating maintenance and transportability. The remaining KAPSE modules are written in host-dependent Ada and, except for those which encapsulate host provided services, are also maintainable in a MAPSE.

- The MAPSE architecture provides for a comfortable marriage to a variety of popular host computers and operating systems. These include, of course, the IBM 370 with VM and the Perkin Elmer 8/32 with OS/32. In addition, there is a natural mapping to such extreme hosts as a bare machine (without operating system) and a distributed network of heterogeneous computers. The IBM 370 mapping to a set of virtual 370's communicating via VMCF (Virtual Machine Communication Facility) is an example of such a network implementation.

- The use of a combined hierarchical and relational data-base structure, the concept of a database object's distinguishing attributes, and the use of windows and

capacities provides mechanisms for efficient configuration, partition, and project management without the use of cumbersome management tools.

- The choice of Diana, the leading candidate for a common intermediate representation for Ada, provides the proper representation for a state-of-the-art compiler and provides a basis for the development of future intermediate representation based tools.

- The Virtual Memory Methodology (VMM) system is a proven technology for defining, creating, and accessing representations of complex data structures. Its use as a basis for the Diana implementation provides for convenient and efficient mappings to a variety of underlying host storage facilities. Both VMM and the Diana implementation are available as tool components for MAPSE programmers.

- The Ada compilation strategy centers around an Ada Program Library concept which is fully integrated and supported by the KAPSE database. Using VMM, libraries may maintain source, Diana, and object representations of perhaps multiple versions of compilation units, as well as linked executable versions of programs. The compilation strategy, fully supporting Ada, significantly decreases the potentially high cost of the Ada recompilation rules.

- The compiler itself uses well-understood state-of-the-art technology to produce efficient object code while maintaining acceptable compiler performance. User control is provided over the tradeoffs of object code efficiency versus compiler speed, and object code efficiency versus debugging ease. The user may optionally select among different efficient run-time models, tailored for particular applications. In particular, the "static" model, appropriate for embedded systems with limited memory, reduces run-time storage allocation costs to zero.

- The compiler architecture depends on a standard tree-structured representation of Ada (viz., Diana) and the use of a program transformation model of compilation. This model, used systematically throughout the compiler, depends upon the application of program representation transformations, stored in VMM managed tables, to perform the bulk of the compilation process. As a result, the compiler may be easily maintained and tuned. In addition, the transformation-based code select phases both produce very efficient code and may be retargeted in a straightforward way. The tree-transformation facilities are available as tool

components for MAPSE programmers.

- The MAPSE Command Language (MCL) provides a simple, easy to use interactive user interface which, at the same time, provides experienced users a full session-control programming language with powerful features. The Command Processor (CP), which implements the MCL, provides an "interactive tool shell" as a tool component which facilitates user construction of sophisticated interactive programs.

- The source level debugger uses the interactive tool shell to provide access to an executing program, and to information about the program stored in the program library. Each program runs within an "execution harness" which provides a set of "execution control points". Each execution control point, implemented efficiently, represents a place where an external program may seize control. Via compiler switch, a user may modify the granularity of placement of execution control points. The separation of the program and its execution harness from the controlling debugger allows for future controlling programs such as functional simulators, remote debuggers, etc.

- The editor provides a simple access to a powerful set of underlying facilities. Initially a character-based editor, the editor can be extended to allow editing of structured objects. With its clean terminal interface, the editor functions as smoothly with a keyboard-printer as it does with a sophisticated display.

- The flexible KAPSE database facilities allow for a wide variety of configuration and project management styles, from an "open tool box" model for small, unstructured projects, to comprehensive support for multi-version, multi-revision software products, such as the MAPSE itself. By considering from the beginning the use of the MAPSE to develop, test, maintain, and deliver itself, the design is forced to satisfy the twin requirements of usability and open endedness.

Usability

The MAPSE as delivered to the Air Force will be immediately usable for the development of Embedded Computer Software for the IBM 370 and Perkin Elmer 8/32 computers. Indeed, as pointed out in the MAPSE Generation and Support Specification and the Computer Program Development Plan, the MAPSE will be used, following an initial bootstrap, to support its own development, maintenance, and evolution to a full AIE. Such "immersion", the use of software to support its own development, has been shown to

result in a final product which is more robust and reliable than would have resulted without immersion.

In order to be usable, the MAPSE provides a comfortable user view of a simple, not unfamiliar, system built utilizing a small number of concepts. The command language is powerful enough for the most sophisticated programmer or MAPSE operator, yet easy to learn and natural for management use. The most important tool in the MAPSE is the compiler; it generates efficient code to support both Ada application requirements and the MAPSE itself, which is built in Ada. A design problem is to reconcile the need for efficient code with the need for efficient compilation, both in terms of statements per minute and in terms of avoiding unnecessary Ada-required recompilation of dependent units. The remaining tools, especially the debugger, provide the capabilities that knowledgeable users have come to expect. The entire MAPSE is a production quality software system, providing reliable service with minimal system overhead.

Open Endedness

The MAPSE is open ended in order to support both short term extension to an APSE and longer term extension to an AIE. Short term open-endedness, providing an Ada Programming Support Environment, is accomplished by including facilities for portability, tool introduction, manager-controlled management, and APSE administration.

Short Term.

MAPSE portability potentially involves both the retargeting of the compiler and the rehosting of the entire APSE. The compiler is built in a manner which isolates target-dependent portions and facilitates their systematic parameterization. A future operational APSE may well simultaneously support Ada compilers for several different targets. Tool rehosting is reasonably straightforward: tools are coded in "target independent" Ada, with all target-dependent services provided by the KAPSE. (Note that Ada is a machine-independent language in which one can write machine-dependent programs; one must be careful in order to write target-independent Ada.) Thus tools need only be recompiled in order to be moved. Rehosting of the KAPSE itself is not as straightforward. The KAPSE, although coded in Ada, is clearly host dependent. Indeed, KAPSE efficiency comes in part from judicious use of substrate operating system and file system facilities. KAPSE rehosting, then, requires both a reorganization (and, perhaps, a partial reimplementation) as a result of new host dependencies, plus a remapping of the KAPSE on a new operating system substrate.

An approach to tool construction is the "chip set" model, which provides the beginning of a conceptual basis for a software parts technology. Tool granules, analogous to integrated circuit chips, are categorized into "logic families" which have well defined packaging conventions and standardized "data in", "data out", and "control" lines. The initial MAPSE tool functions are not provided by monolithic tools; rather, they are provided by appropriate compositions of tool chips. New tools are created by constructing new tool chips if necessary, and composing them with existing tool chips by means of the tool-KAPSE virtual interface, providing new tool functions. Examples of tool chips to be described later include the lexical analyzer, the parser, the pattern-replacement transformation facility, and the interactive tool shell.

The APSE approach to project and configuration management is that "managers should manage". There is no built-in, system mandated management style. Rather, a manager selects an appropriate management framework in which work assignments may be carried out in a controlled manner. Alternatively, a manager may develop a framework specially tailored to the problem at hand. Management frameworks for different management styles, from the "open toolbox" model, through a full model for the support of a multiple-version, multiple-revision product, are provided. Thus, the APSE conveniently supports large, long-lived projects as well as small, short-lived ones.

Stoneman barely hints at the problems of APSE operation and administration, yet these problems are considerable. An APSE does not remain static: projects come and go, underlying hardware changes occur, systems crash at inconvenient times. APSE administration facilities include the ability to define new categories of database objects with their associated attributes and operations, to reallocate system resources as appropriate, and to authorize new instances of the various classes of users.

Long Term.

In the longest-term extension of a MAPSE, a complete Ada Integrated Environment will deal with issues beyond the direct support of the programming process. Support for the complete software life cycle requires, for example, tools for generating, modifying, and verifying requirements and specifications, and this, in turn, requires requirement and specification languages. This causes an AIE to be potentially multi-lingual. Full support for ECS development requires some form of on-target debugging, which in turn requires a (potentially) distributed AIE, perhaps utilizing either a network or some form of detachable tool. The very rapid development of support architectures and programmer workstations requires the ability to consider a network of personal computers as a possible AIE host. Similarly, the rapid developments in the field of office automation requires that such

facilities be made available either within the AIE, or by means of a gateway to an office information system. It is worth observing that programming is an activity which typically is performed in an office. Thus, the facilities of an office automation system most certainly will be of benefit to programmers. A future AIE will provide support for the many different types of work centers associated with a software project, such as documenting, accounting, training, and contracting, as well as programming and managing.

Future AIEs will provide support for a variety of different types of work centers, patterns of coordination and cooperation among such centers, and facilities for reorganization as a result of organizational change or of technology insertion. Consequently, current MAPSE development must be viewed in the context of providing a base for such future systems. Our MAPSE design does provide such a base.

STRUCTURE OF THE MAPSE

Chapter 2 presented a preview of the architecture of the MAPSE, starting with the Stoneman ring diagram (Figure 2-1) illustrating the hierarchy of level 0, KAPSE, MAPSE, and APSE, with added emphasis for lines of separation and interface. This figure was elaborated (Figure 2-2) in order to make explicit the user-MAPSE interface, and the provision of run-time Ada support for executing Ada programs. Figure 3-1 re-presents Figure 2-2, noting specifically

- the database portion of the KAPSE,
- the host dependent implementation of portions of the KAPSE,
- the necessity of separate run-time support for each Ada program, and
- the possibility that separate Ada programs might, in fact, run on separate hosts.

The Host-Independent Structure

If we look more closely at any one Ada program, be it user program, tool, or the database, we see that it consists of layers: the run-time system, the program itself, and the encapsulating KAPSE layer (Figure 3-2). The interface between KAPSE and program is the SOW-mandated virtual interface. This interface, defined by Ada package specifications, provides access to those KAPSE-defined interprogram communication, control, and database access facilities actually used by the program. These pieces of the KAPSE are linked with the program in order to produce the "enKAPSElating" layer. The host, not explicitly

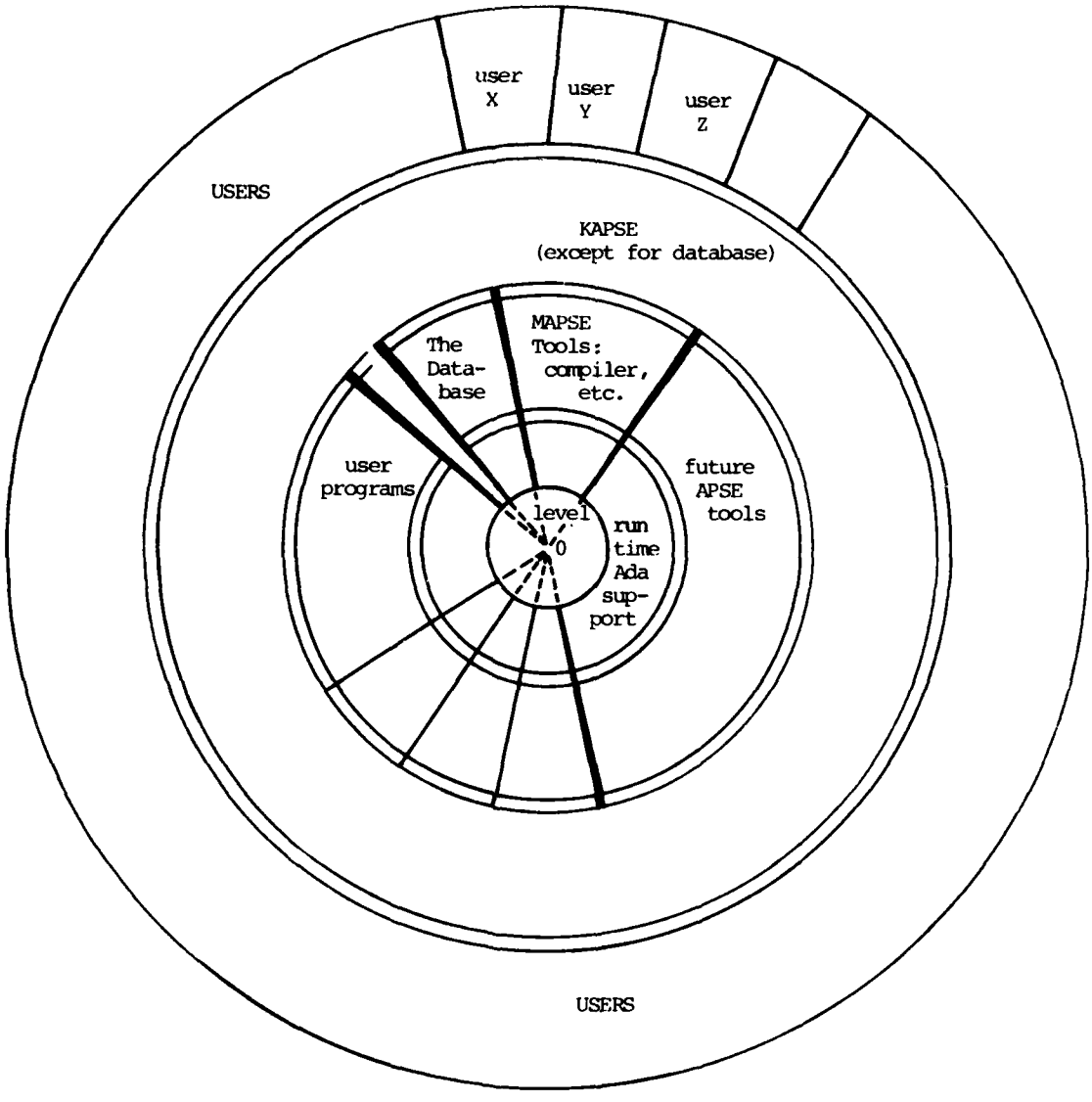


Figure 3-1

visible in this figure, provides support for the run-time system and for portions of the KAPSE. The run-time system provides an "Ada virtual machine" which provides support for such language facilities as tasking, timing, exceptions, and storage management. Various run-time models may be selected by the user at compile time.

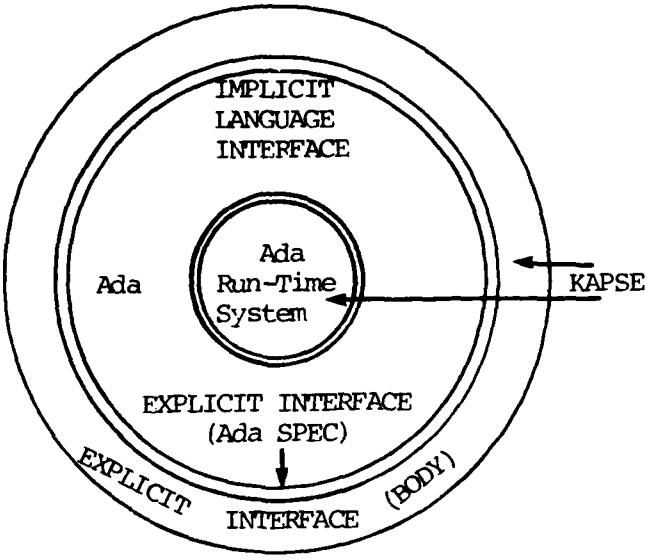


Figure 3-2

With the structure of a single program in mind, we can redraw Figure 3-1 at some instant in time, Figure 3-3, when

- User A is typing a command,
- User B is running the compiler,
- User C, in the middle of editing, has invoked a database inspection program called LIST_NAMES,
- User D is running two tools, with the output of one feeding the input of another, and
- User E is debugging an ECS application.

Figure 3-3 shows a variety of relationships between cooperating Ada programs:

- A's Command Processor is unrelated to B's Command Processor. They are unsynchronized unless they should attempt to access a common database object.
- C's Command Processor has invoked the Editor. The invoking program is suspended until the invoked program completes.

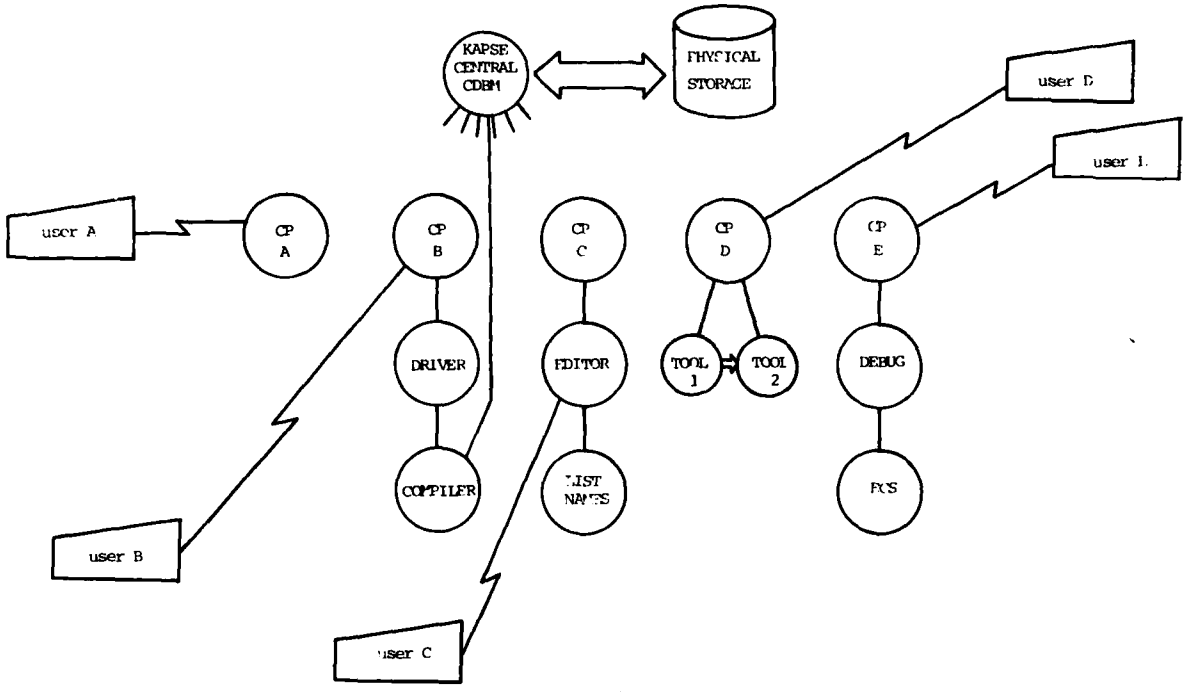


Figure 3-3

- B's Command Processor has <u>created</u> a background job, which executes concurrently with the creating program. D's Command Processor has created a co-routined execution of two strongly interacting tools.

- E's Debugger is <u>controlling</u> an ECS application program. A controlling program may start, stop, or modify a controlled program.

- B's Compiler has <u>requested</u> service from the KAPSE/database in order to provide file access. To avoid cluttering the figure, all such requests have not been shown. In fact, terminal input/output also appears as request for service from the database.

An important MAPSE feature not illustrated in Figure 3-3 is import/export communications with the environs of the MAPSE. Such communication is essential in order to allow delivery of releases of MAPSE-produced software, either ECS software or new revisions of the MAPSE itself, and transfer of software between MAPSEs on different machines.

Thus, the overall structure of the MAPSE can be seen to consist of a collection of cooperating Ada programs: the

database, tools, user programs, and ECS applications under development. This cooperation is mediated by the KAPSE which provides a host-independent interface supporting multiple Ada programs and a host-dependent interface whose character is determined by the mapping of the KAPSE to the underlying host plus operating system. The single shared database is accessed via the MAPSE interface package in the same manner as other KAPSE services are accessed. Provision of a single database provides for synchronization, enhances security, improves efficiency, and simplifies the problem of mapping the database onto the underlying host operating and filing systems.

The Host-Dependent Structure

The MAPSE depends, in all respects, on the quality of the KAPSE, as visible at the KAPSE-MAPSE interface, and as invisible in the KAPSE implementation. As seen at the KAPSE-MAPSE interface, the KAPSE provides the services of a portable operating system and database designed to support multiple cooperating Ada programs. Such a KAPSE could be implemented directly on a bare machine host or on top of an existing operating system. A bare machine implementation avoids the problems of mating with nonstandard operating systems and allows a KAPSE with no host-specific idiosyncrasies. However, a bare machine implementation requires more software to be written than does a host operating system implementation, and, more seriously, requires that the host machine be dedicated to the running of the MAPSE. Although perhaps acceptable for a machine like the Perkin Elmer 8/32, this is unlikely to be acceptable for an IBM 3033. An implementation on top of a host operating system allows simultaneous non-MAPSE use of the host, use of the host file system to support the database and import/ export functions, and avoids the necessity of writing code to perform time-sharing, address space management, etc.

The design of a MAPSE must be considered as two parts, the MAPSE toolset architecture, and the KAPSE architecture. The toolset architecture is discussed at length in the next section. Briefly, it involves the coding of tool components in a host-independent subset of Ada, interfacing these components with each other according to MAPSE-standard conventions, and subjecting the resulting tools to appropriate MAPSE-standard management and control.

The architecture of the KAPSE depends crucially on the successful mating of the KAPSE to its substrate host. The S.O.W. specifically identifies two host systems: The IBM 370 with the VM operating system, and the Perkin Elmer 8/32 with the OS/32 operating system. We feel, however, that it would be quite naive to consider just those two systems. Rather, we have considered those two, a variety of conventional machine-operating system pairs, bare machine implementation, and network implementation.

Our architecture is, at the same time, optimal for both the 370-VM and 8/32-OS/32 pairs and efficiently implementable on these other hosts.

The KAPSE architecture is such that services which are provided by a host may be used directly or indirectly to implement internal KAPSE package specifications. As a simple example, the host operating system clock is used to implement the KAPSE time facilities. Since the KAPSE architecture does not assume a host operating system, it can be implemented on a bare machine. Since the KAPSE architecture anticipates a variety of specific hosts, there is a natural mapping of the KAPSE onto those hosts.

Figure 3-4 shows the mapping of the MAPSE onto a bare machine. In this case a full KAPSE is implemented, including a complete time sharing multiprogramming monitor, the low-level operating system support routines, a file system, and the underlying device control functions. Although such an implementation is possible, it is not desirable for the reasons previously mentioned.

Figure 3-5 shows the mapping of the MAPSE onto the Perkin Elmer OS/32 operating system. In this implementation each Ada program is run as a separate OS/32 task. Note that each Ada program, in turn, has its own Ada run-time system, including support for its own Ada tasks. OS/32 task functions are utilized to implement time slicing and management, interrupt handling, separate address spaces with shared segments, and generalized inter-task communication. The OS/32 file system supports import/export, provides database storage, and an OS/32 task load module. This implementation is the best choice since it is done at lowest cost and highest efficiency because of OS/32 utilization, and allows concurrent use of MTM and other non-MAPSE activity.

The MAPSE implementation on VM is more subtle. The initial observation is that, since VM provides a user a virtual 370, a bare machine implementation on such a virtual machine should be used. This approach requires the implementation of the "bare machine" functions previously mentioned and requires allocation of the entire MAPSE to a 16 megabyte address space. More seriously, a multi-access, secondary virtual storage operating system does not perform well on VM/370. Indeed, IBM itself discourages the use of TSO on VM. Thus a "bare machine" implementation on a single VM is a poor choice resulting in high cost and complexity, bad performance, and poor utilization of the underlying VM/370 system.

A second approach to a VM implementation might allocate a single VM to each Ada program. This approach, although initially attractive, contains a fatal flaw: under CP, one virtual machine cannot create another virtual machine. Thus, new Ada programs

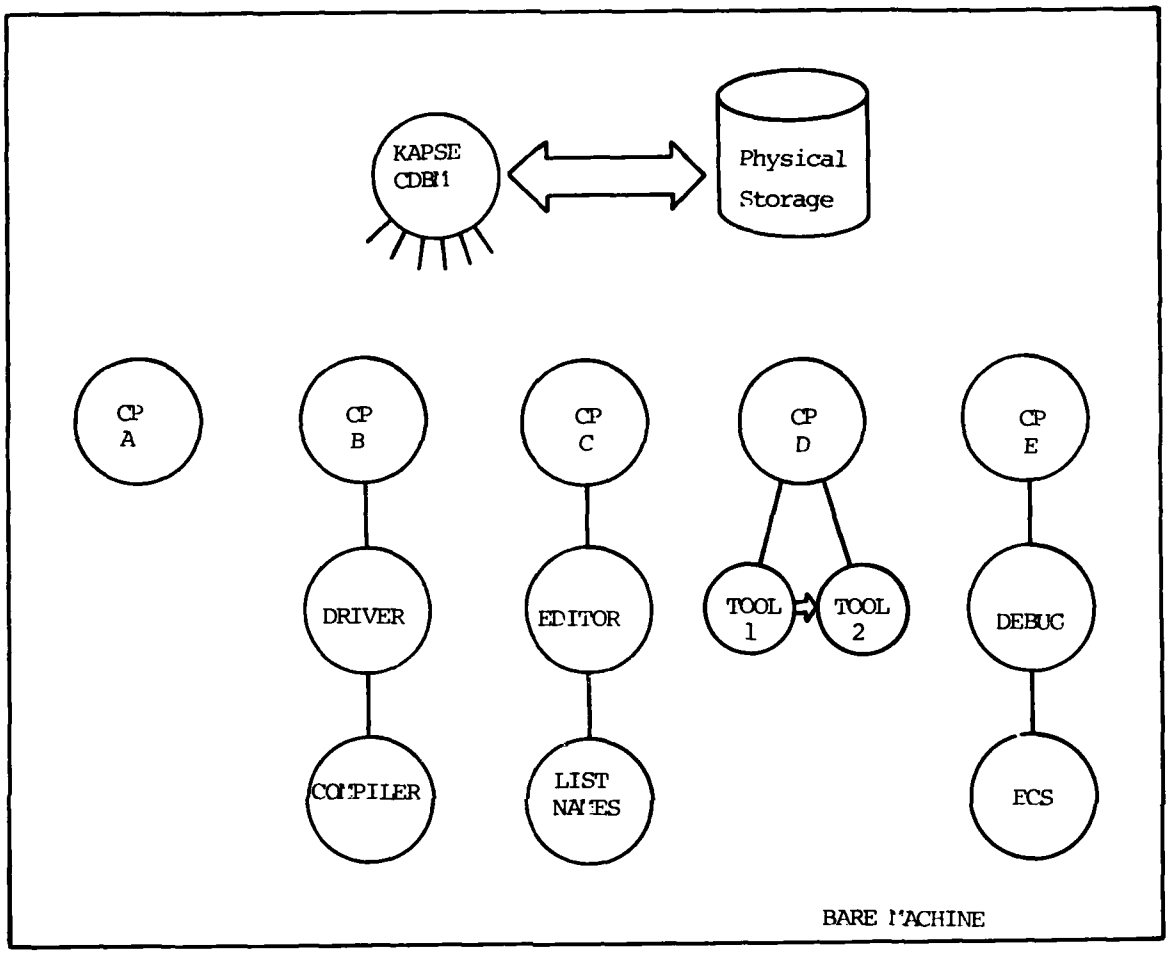


Figure 3-4

cannot conveniently be created.

Figure 3-6 represents the proper architecture, one virtual machine for each MAPSE user, with the database manager being considered a user. Thus, a new virtual machine need only be created when a new user is introduced to the system, which is, in fact, the VM/370 convention. This approach allows only the database virtual machine access to database devices reducing the VM/370 physical input/output "bottleneck", allows an ample 16 megabyte address space for each user's multiple Ada programs, uses CP functions to support time sharing and user response, and utilizes the Virtual Machine Communications Facility (VMCF) as a high speed memory-to-memory inter-program communications

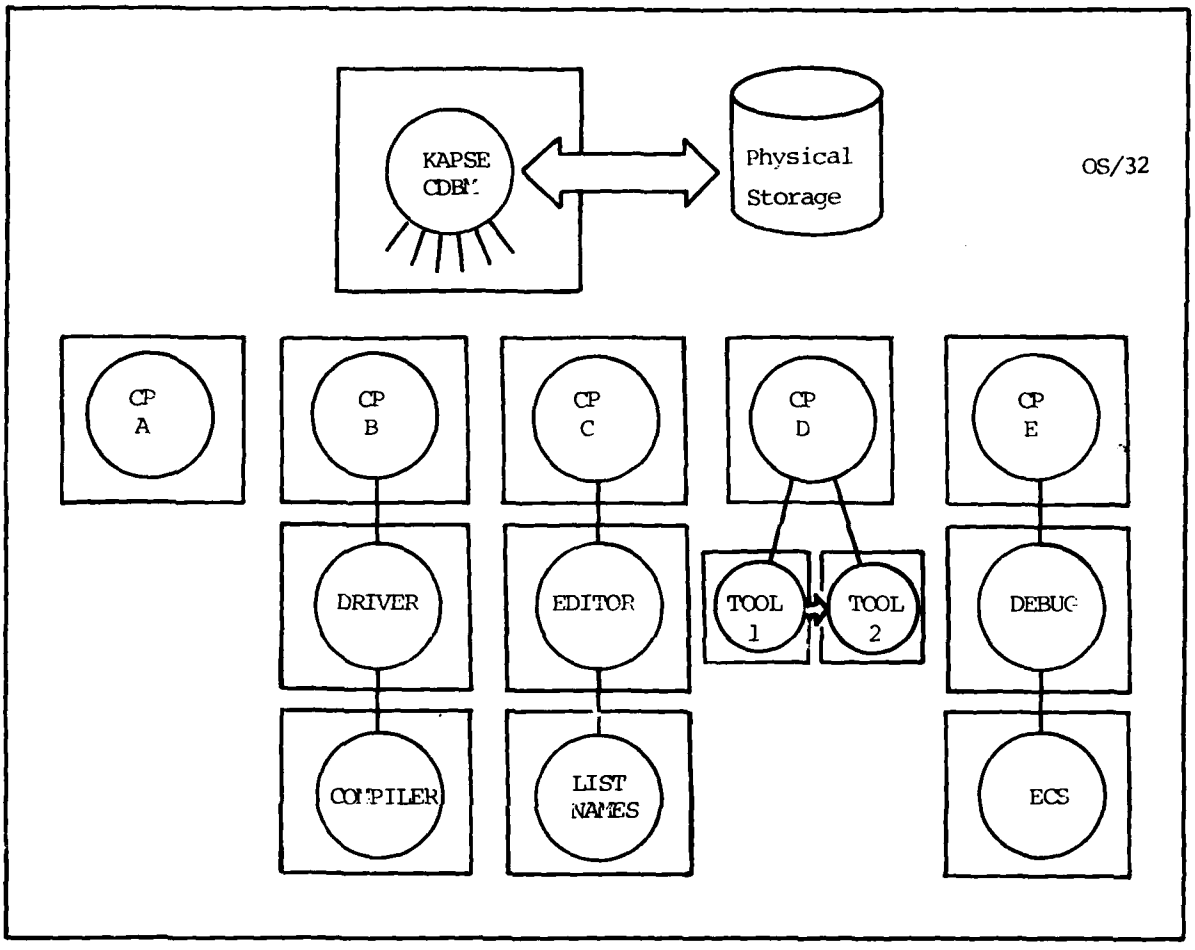


Figure 3-5

protocol.

We are confident that this MAPSE design can be implemented in a straightforward manner on such modern machine architectures as local networks of personal machines. The 370 implementation is, in fact, such an implementation: the host is a network of virtual 370s, utilizing VMCF as the network protocol.

TOOLSET ARCHITECTURE

The MAPSE will be extended to an APSE by the addition of tools. In this section we examine those characteristics of the MAPSE which allow for the orderly and cost effective addition of

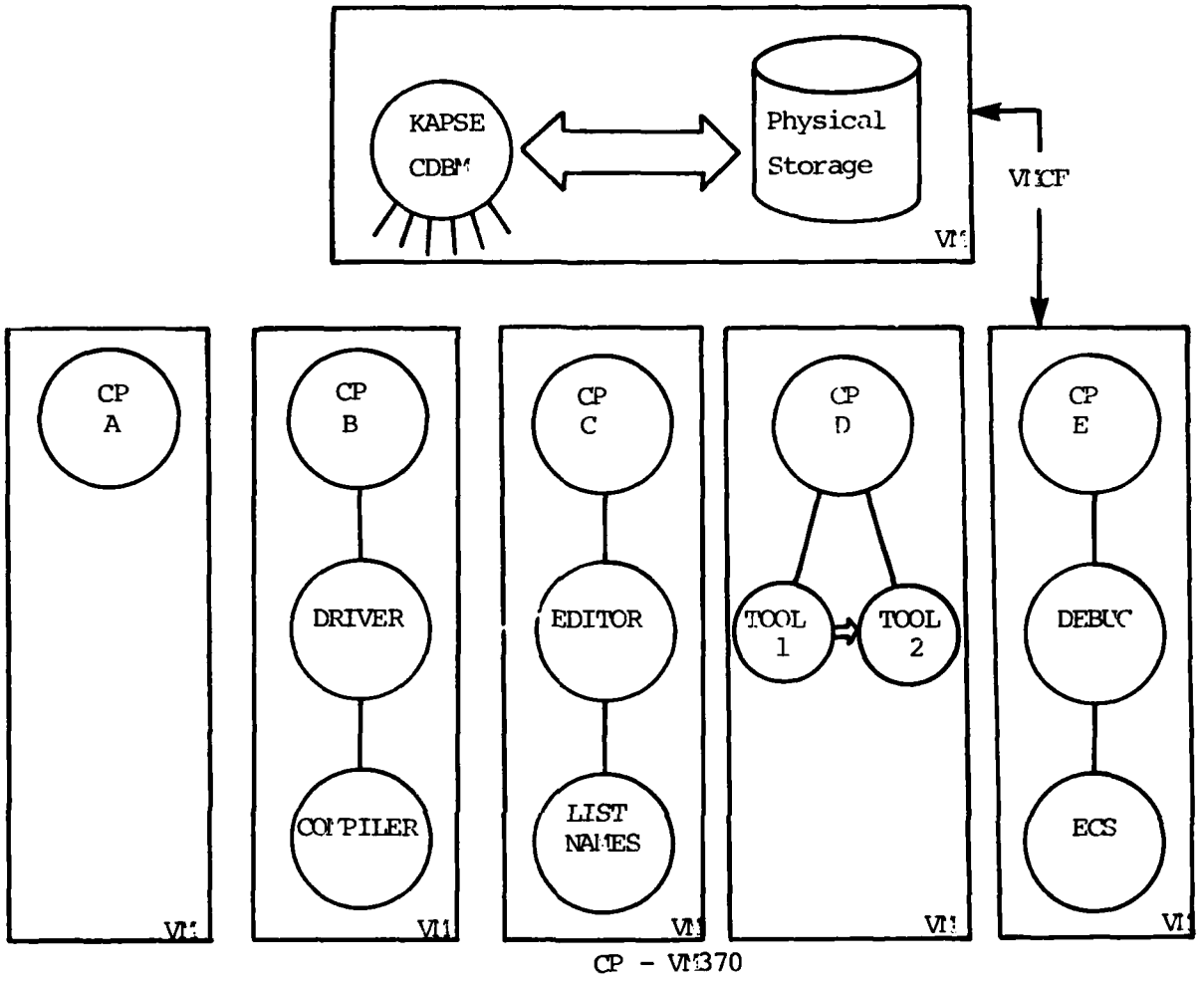


Figure 3-6

tools.

Before proceeding, it is important to define "tool". We use tool here not in the sense of an isolated, individually-owned, hand-held tool, but in the sense of part of the tooling needed to carry out a cooperative enterprise. Tools are the product of tool engineering "a branch of engineering in industry whose function is to plan the processes of manufacture, develop the tools and machines, and integrate the facilities required for producing particular products with minimal expenditure of time, labor, and materials." (Webster's Seventh New Collegiate Dictionary) Our emphasis here is on the development, integration, distribution, and management of cost-effective software tools.

Tools, as we have defined them, are not operated in isolation but interact with other tools. In order to ensure the orderly development and accurate functioning of interacting tools, tool interfaces must be developed, distributed, and managed in much the same manner that tools themselves are. Well-designed, well-managed tool interfaces ensure interoperability of tools. Tool interfaces serve the same functions as jigs and fixtures in manufacturing, which "set the relationship between the work and the machine tool" [Sedlik 70] and in so doing:

"1) Ensure the interchangeability and accuracy of parts manufactured,

2) Minimize the possibility of human error,

3) Permit the use of unskilled labor, and

4) Reduce manufacturing time." [Sedlik 70]

Software tool development has often been costly due to the limited ability to re-use existing tools and tool pieces. The following tool characteristics reduce both initial tool cost and support costs:

- Availability as a ready-to-use package from an organization which will support it.

- Portability, so that the same tool may be used on different hardware and in different operating environments.

- Adaptability to different users so that the same tool may support different operations. For example, a text editor may be used both for programs and documentation.

- Modifiability as the need for extensions is recognized.

- Composability, so that several tools or tool pieces can be easily combined to form new tools. For example, a text editor and a sort program can be combined to provide a concordance generator tool. See [Kernighan 76] for other examples.

As the preceding list illustrates, a cost-effective tool has reasonable initial cost, does not easily become obsolete, and spawns related tools cheaply - such related tools share development costs, have complementary human engineering, and represent a smaller body of distinct code for a support organization to maintain.

Tool interfaces which are well defined and well managed encourage tool use and tool adaptation. Adaptaton at the level of the tool interface can accomplish:

- Compatibility with other tools and data structures.
- Provision of standard user interfaces.
- Specialization of tool functions.
- Pre- and post-processing of inputs and outputs.
- Encapsulation to filter out bad inputs, provide help facilities, usage logging, etc.
- Extension of functional capabilities.
- Human engineering for higher or lower skill levels.
- Compatibility with different hardware and operating environments (e.g., interactive graphics vs. text batch).

Not only will there be many installations of the MAPSE, but we can expect that these installations will in time be networked both to one another and to other non-APSE equipment. In particular, we expect to see gateways between APSEs and both existing networks and office automation systems. The tools which are built on one MAPSE should be easily portable to another MAPSE. Moreover, tools on interconnected APSEs should be able to interact as easily as tools on a single MAPSE via a common tool interface. Lastly, the interface adaptation mechanisms should accommodate the interconnection of APSE-based tools to "foreign" systems, and to tools written in "foreign" languages (e.g., JOVIAL).

A tool will be provided to its users by a tool distribution organization or a "tool distributor" which is responsible for the tool. This organization could be responsible for tool development and maintenance but would more commonly interact with separate tool development and maintenance organizations. The tool distributor provides not only the tool but whatever documentation, help facilities, usage and fault logging is appropriate. The distributor may withdraw the tool from distribution or "recall" faulty tool models. These functions of the distributor cannot be reliably accomplished unless a link is maintained between the tool and the distributor. This link might be maintained by keeping "a file of filled out warranty cards" or by providing each tool with a distributor interface. This interface would be invisible to an ordinary tool user but would provide two-way communication between the tool and its distributor for fault logging, update of help facilities, etc.

It is clear that any given tool may have many interfaces. It is essential that these interfaces be specified separately from one another for maximum composability and adaptability. For example, it may be desirable to log the error messages produced by the compiler. If the error message interface is "buried" in a generalized text output interface it will be difficult to build the error logger. Moreover, the error-logger builder would then have to know more about other compiler text outputs than is necessary.

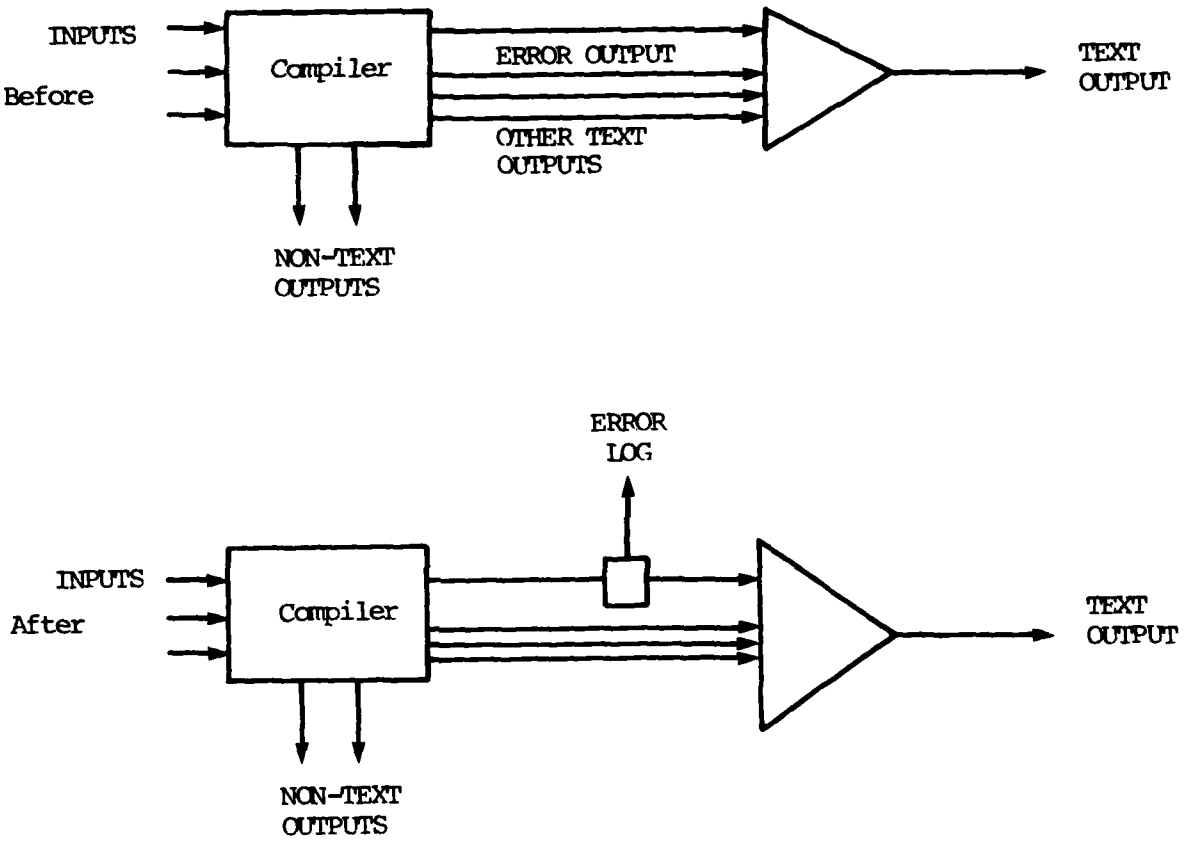


Figure 3-7

Figure 3-7 suggests the diagrams which show the interconnections between integrated circuit chips. Electronic hardware costs have come down because engineers can design the interconnections between a small number of chips rather than designing circuits of a much larger number of discrete components. This is successful because families of chips with standard logic levels, rise and fall times, etc., may be reliably

interconnected with one another. In addition, there are chip sets which are meant to complement one another in the building of related applications.

A software "chip set" architecture is beginning to emerge which allows the reliable interconnection of software pieces. The tools of the MAPSE form such a chip set.

Definition of 'Tool'

In one sense there is nothing special about a tool in the MAPSE. All Ada programs, be they user programs or tools, are invoked from other Ada programs or tools by means of the same KAPSE mechanism. Similarly, there is no distinction in mechanism between user command language scripts and command language scripts which act as tools. The special nature of a tool, or an Ada fragment which can serve as a tool component, is its management control. Tools and tool components are written according to MAPSE-standard tool implementation conventions and are subjected to MAPSE-standard management and control.

Since Ada is an implementation independent language in which one can write implementation dependent programs, MAPSE tools and tool components must be written in a subset of Ada which avoids implementation dependencies. The result of this discipline is that tools and tool components are portable simply by recompilation. Tools must be written with a careful use of default input and output in order to facilitate coroutining and input-output redirection by the Command Processor. Each tool must be provided with a text "help file" in order to utilize the Command Language HELP command. The tool must be stored in an appropriately mnemonically named database object stored in an appropriate tool portion of the database. Tools must be parametrized in order to allow suppression of non tool-invariant output, such as time reporting, version numbers, execution statistics, and representations of internal names, in order to allow semiautomated testing by means of output comparison. These tool management issues will be discussed in more detail later.

Following the "chip set" model, tools may be interconnected in a variety of ways. Examples of coroutined interconnection are given in Chapter 6. Of particular concern are the reusable tool components which facilitate the construction of new tools. These are presented in the next section.

Tool Components

The tools provided in the MAPSE are spelled out in the S.O.W. as Editor, Compiler, Linker, Debugger, and Command Language Processor. These are all described in detail in subsequent chapters. A variety of management facilities, such as

configuration management, are also identified in the S.O.W. and described in subsequent chapters. Of interest here is the S.O.W. requirement that

> "...Software performing a single function required (or potentially required) by more than one system component shall be designed to be reusable to the maximum extent possible..."
>
> [S.O.W., 4.1.1.5]

Although each of the modules within each tool, as a result of the tool coding standards, meets this requirement, certain modules, which we call "tool components" are of particular interest. The major tool components, along with brief descriptions, are:

- Lexical Analyzer - a finite state machine Ada lexical analyzer.
- LEXSYN - An LR based Ada syntax analyzer which produces an Abstract Syntax Tree (AST) program representation.
- SEM - An Ada semantic analyzer which transforms an AST into a Diana program representation.
- Diana - An Ada package which defines and implements the Diana program representation abstract data type.
- Pattern Matcher - a table driven pattern-replacement transformation facility for Diana program representations.
- Execution Harness - A generalized mechanism for program control.
- VMM - Virtual Memory Methodology, a generalized access method used to implement Diana, as well as other system components.
- Interactive Tool Shell - A general purpose interactive user interface derived from the Command Processor.

The Lexical Analyzer, LEXSYN, and SEM are described in Chapter 8, as is Diana, which has its own rationale in its reference manual. The Pattern Matcher, Execution Harness, VMM, and Interactive Tool Shell are described below.

The Pattern Matcher.

The Compiler utilizes Diana as a standard tree-structured representation of Ada compilation units, and depends on a Diana-

to-Diana transformation model of compilation. The advantages of this architecture are described in Chapter 8. Transformations on the Diana representation are performed by the use of a general purpose pattern matcher which is driven by tables of stored Diana transformations. These tables are systematically maintained by the Virtual Memory Methodology system.

Over the past several years, there has been a growing interest in the so-called "source-to-source transformation" paradigm [Loveman 77] for program refinement, development, and compilation. Although not a part of this MAPSE effort, the use of Diana representation and the availability of the Pattern Matcher will facilitate the application of this ongoing work to Ada as the MAPSE evolves to an AIE.

The Execution Harness.

The Execution Harness is not a tool component, rather it is an Ada program control situation which results from the cooperation between the compiler and the KAPSE. At compilation a set of "execution control points" is identified. Each execution control point represents a place in the program where an external program may seize control. For example, the Debugger places breakpoints at execution control points. At an execution control point a controlling program may stop, modify, or start a controlled program. The idea of execution control points allows separation of the concept of program control from the concept of debugging. Although in the MAPSE the only controlling program is the debugger, in the future there will certainly be function simulators, remote debuggers, and other forms of controlling programs.

Functional simulation provides that, at each execution control point, a functional simulation execution monitor gains control of the running program. The monitor provides a simulated hardware environment for the running program and will perform actions such as advancing a "pseudo clock" based on anticipated target machine characteristics, simulating Ada tasking and timing in terms of the target environment, providing simulated target input conditions, and updating the external model of the target environment.

Future development of the MAPSE will certainly involve, for example, ECS software development for microprocessors. Such programs, if written following the tool coding conventions, may be debugged in the MAPSE. They may then perhaps be run under control of a functional simulator. At some point, however, they will have to be executed on their appropriate microprocessor. This will require, at least, the retargeting of the Compiler for that microprocessor. Our design, however, allows more. Assume that a microprocessor development system is attached to the host machine. A network-like implementation of the MAPSE will allow

programs under MAPSE control to be executed on the development system, while the execution control point concept will allow the debugger to interface with and debug the running program on the development system.

Virtual Memory Methodology System.

MAPSE tools must be able to preserve data structures within the database in order to communicate with other tools or with subsequent activations of the same tool. In general, it cannot be assumed that the address space of the host machine will be adequate to keep such data structures entirely within memory while they are used. The Virtual Memory Methodology (VMM) system provides both a means of representing the data structures used by tools in a consistent and efficiently-accessed external form, and a means of overcoming address space limitations on the size of data structures. In addition, VMM provides aids to debugging and communication between hosts.

The VMM system is a technique for defining, creating, and accessing representations of data structures. The use of the VMM system to implement a data structure in Ada provides the following capabilities:

- A permanent, directly accessible representation of an instance of a data structure may be created in the KAPSE database that can be efficiently accessed by any MAPSE tool which uses the same definition of the data structure.

- Since software memory management is part of the access method, the direct addressability of such data structures is independent of the actual addressing range of the host system.

- It is possible to perform automatic conversions between a directly accessible representation of a data structure and either of two host system independent linear representations:

    - a human-readable text, which is primarily of use for debugging and testing, and

    - a compressed binary form, which may be used to transfer a representation between hosts. While the human-readable form could also be used for the latter purpose, compressed binary is much more compact and uses fewer resources for the conversion process.

Interactive Tool Shell.

Figure 3-8 represents the overall architecture of an arbitrary interactive tool. The central component, the Interactive Tool Shell, provides complete support for a user command environment including Command Language interpretation, processing of filed scripts which are parameterized sequences of Command Language statements, input-output control including piping and arbitrary redirection, and session record maintenance. As the Interactive Tool Shell is used in the implementation of the Command Processor, Chapter 6 contains details of its implementation and the interactive language which it processes.

The shell assumes line-at-at-time input and produces line-at-at-time output. A user keyboard device appears, as a result of the low-level KAPSE terminal input mechanism, as a data base object of a category appropriate to the particular keyboard. Naturally, any facilities which are built in to the keyboard such as local editing, are available to the user. The database object which represents the keyboard is processed by the line editor pseudo terminal interface. This line editor implements system-wide local line editing conventions including erase character and erase line functions. Separation of the line editor from the physical keyboard allows arbitrary text data base objects, of the proper category, to serve as substitute input devices. Similar considerations apply to output. Thus, the Shell supports both batch and interactive input-output, and is extendable to support display input-output devices. Indeed, since the shell is just an Ada program a user may potentially have several shells running. This requires a bit of care as the user slews from one shell to another, but can in the future be supported conveniently by a display system allowing multiple windows, one for each shell, presented side-by-side on one display, on several displays, or overlaid in whole or in part. This ability to support asynchronous activity at the user level naturally allows, for example, a compilation to proceed in parallel with an editing session in a manner which is invisible to the tools.

The Interactive Tool Shell maintains, as a data base object, a session record. This record contains the interpretive context of command language evaluation, as well as references to all currently visible data base objects. Thus, the saving of the session record allows a user to suspend a session and return to it at a later time.

Particular interactive tools such as the editor and debugger have their own application specific tool functions which are incorporated into the framework provided by the Shell. In addition, the Shell provides the ability to run an arbitrary Ada program. This is the mechanism by which, for example, the Command Processor invokes the Editor. Indeed, the Command Processor may invoke itself, recursively, providing dynamically scoped work sessions. The Command Language provides a uniform

facility for program invocation, which does not depend on the user knowing whether the program is, in fact, a script, a built-in facility, a MAPSE tool, or an arbitrary program.

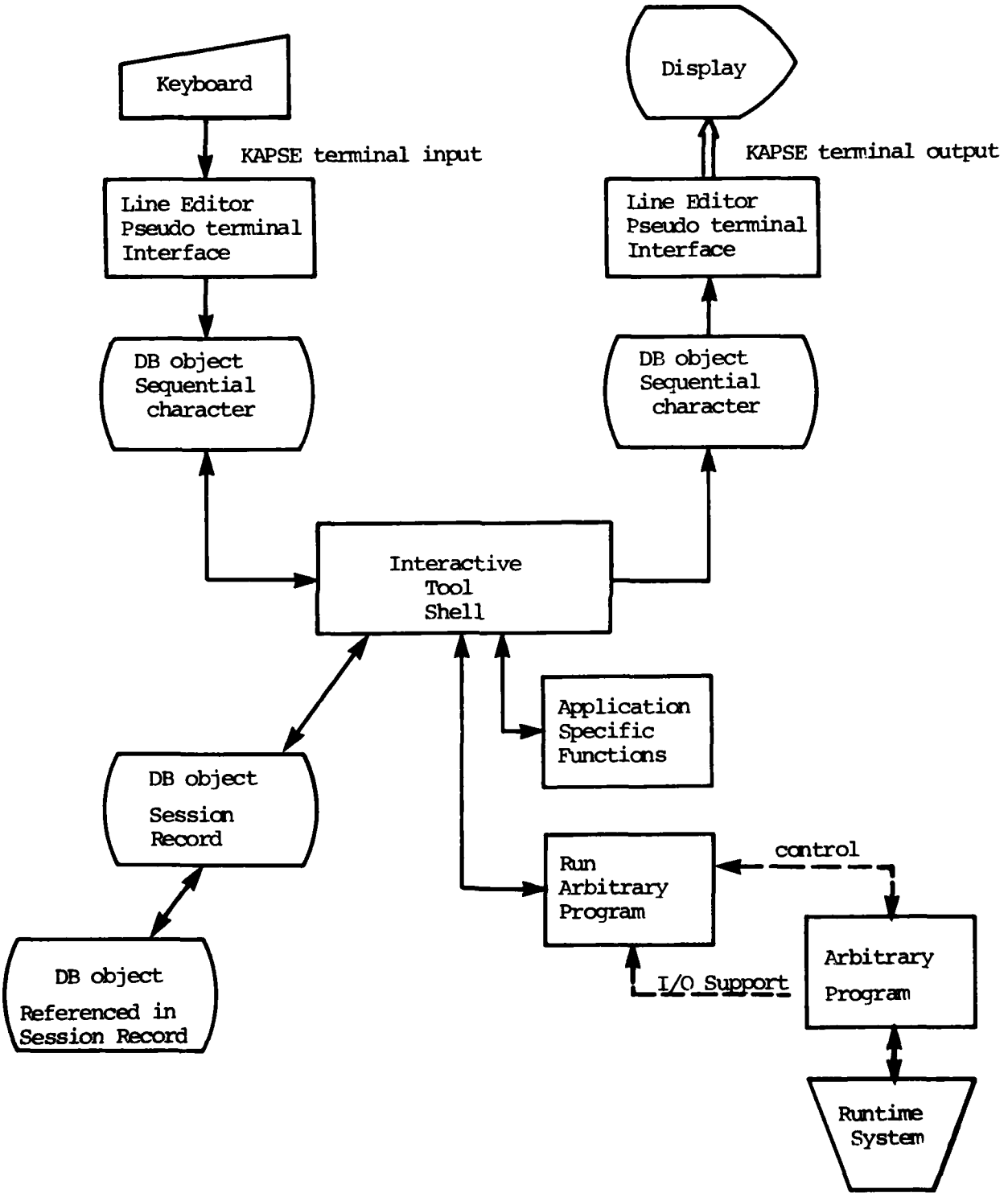


Figure 3-8: An Arbitrary Interactive Tool

TESTING CONSIDERATIONS

In this section, we examine those aspects of the MAPSE architecture which address the testing requirements listed in Chapter 2. Many requirements of testing are directly addressed by sections of STONEMAN and the S.O.W. Certainly, the image of the expanding toolkit - as MAPSE becomes APSE - should include the analytical tools desired for requirements tracing and static debugging [STONEMAN, 7.A.8 and 7.A.10]. In the same vein, our design offers choice among simultaneous, parallel versions of modules and thereby provides a natural means of configuring assorted test environments [S.O.W., 3.2.3 and 3.2.4]. On the other hand, quite a number of test requirements are met by aspects of our particular design from the viewpoint of the more general system which the STONEMAN describes and which the S.O.W. calls out as general design goals for any APSE. As we revisit the test scenarios of Chapter 2, we show how the architectural aspects of our MAPSE design respond to the stated needs. In so doing, we touch on testing needs (less the deferred analytical tools) of elements of the evolving system as they progress from unit testing, through acceptance, and beyond to re-certification after modification.

The first stage in unit test is not application of a debugger, but the configuring of an appropriate test environment. This includes not only stubs (which our Linker generates on request) [S.O.W., 4.1.11.3], but test drivers, and possibly changes to program-internal tables. For example, comprehensive testing of the Virtual Memory Methodology system requires an artificially small set of real memory resources to exercise software paging activities in a reasonable fashion. These needs are directly addressed by system-provided features for selection from multiple versions - in this case a version to exercise the system as opposed to a resource to support an arbitrary Ada program.

Once the module is housed in an appropriate test environment, the MAPSE debugger may be used to control execution, inspect and modify program variables, etc. The rationale of this MAPSE tool is the subject of Chapter 9.

Once one or more such units have been tested, a relatively complete program module can be subject to more extensive testing. Often, one wishes to apply a number of test cases, which are rightly generated at the design-stages of a project. Furthermore, most test stimulus (of MAPSE components, at least) is applied through the default input file. Testing requires automated application of prepared test cases. The need for application of prepared test scripts is addressed by redirection of standard input, taking it from a database object rather than the terminal. The need for (probably repeated) procedural application of a sequence of such cases is provided by a feature of the MAPSE Command Language and its Processor, whose rationale

is the subject of Chapter 6. Scripts may be prepared and applied to the MAPSE Command Processor in order to apply a prepared test suite.

The success of a test is determined by examining results. Recall from our earlier discussion on toolset architecture that tools produce a number of separate results: user visible text and/or binary, as well as system-recorded fault and usage data. All such data must be inspected in order to accept the tested program, and must be retained for retesting in the future. The saving of output is facilitated by redirection of default output to a database object which can be named in the MAPSE Command Language. The results of such program acceptance testing are nicely retained as a composite database object and include input, command script, and accepted outputs for each case.

Once initial testing of a program is successful, we anticipate the retesting that is required. Over the life-cycle of a piece of software, most testing activity is regression testing. Bell Labs developed an automated software testing system in which "regression testing was done using less than 20 percent the machine time of conventional approaches and...can be rerun at any time using clerical personnel" [Jessop 76]. Our design supports such a methodology. Recall that acceptance tests and results were retained as composite objects. These tests can clearly be rerun to form new composite objects, and scripts used for acceptance testing can be augmented to compare the composite objects and to list discrepancies. During this mode of testing, it is necessary to remove test dependent information, such as time of day, from the output. The MAPSE-standard coding style for tools facilitates the removal of this information.

This form of testing presents an attractive way of minimizing the cost of regression testing and thereby, life cycle testing cost as well. It should be improved, however, by eventual addition of a "closed-loop" test driver: one which considers current test results before supplying more script to the MAPSE Command Processor.

Chapter 4

KAPSE

KAPSE APPROACH

The KAPSE and database as described in STONEMAN and required in the S.O.W. are the core elements in the objective to achieve an Ada Integrated Environment. In a sense, the KAPSE/Database is the central connector and controller of a "data bus" onto which all MAPSE tools, as well as users, may be attached.

The design demands placed on the KAPSE are particularly stringent. A very large number of the S.O.W. requirements directly, or indirectly depend upon it. These include: management and configuration control; database access and control; addition of new tools; portability in terms of maximum use of Ada; encapsulation of host dependencies and provisions of virtual machine interfaces; efficiency; robustness in the light of human, software or hardware failures, et.al. This would be a tall order for any operating system.

A conventional file and directory system, with separate configuration management tools, an access control table, special or ad hoc interfaces for MAPSE tools, and unbounded storage for history would not suffice and would not meet the S.O.W. requirements for flexible definitions of configurations, partitions, roles, attributes, categories, history, etc.

Our design approach has been to postulate a small set of integrated concepts that would encompass both the stated requirements for a MAPSE, and the long range requirements for a complete Ada Integrated Environment. These concepts center around a description of the database and its objects in terms of "distinguishing attributes." These objects are viewed and controlled through "windows" on the database that convey rights of visibility, control and manipulation.

Because of the importance of these concepts to our design, this chapter presents an extended technical introduction and gives insight into how many of the S.O.W. requirements are satisfied. The central rationale is one of proposing a minimum set of concepts and facilities which will span all explicit and implied requirements. The first half of this document presents the selected approach, which should be evaluated against our assertion of minimal complexity, the STONEMAN and SOW requirements,

and the list of implied requirements below.

One implied requirement may be named "learnability". The system must be usable with a modest start up cost by persons with no more experience than introductory programming training. This implies a need for a very basic core of capabilities sufficient to get started, and for appropriate defaults, help files, etc. The system must also be "learnable" by sophisticated users from other environments. Here the implied requirement is to avoid "alien" concepts and to incorporate the most powerful ideas from such systems as UNIX.

No requirement on the KAPSE/database is of greater importance than reliability. It is essential that software developed in the first year of a 20-year system life-cycle be just as well supported, documented, and accessible as the final version. A reliability of 99.99% is inadequate. This implies rigorous low-level protection/synchronization mechanisms, comprehensive history tracking, a highly integrated approach to database control, management, and configuration control, and active database protection by the KAPSE.

Learnability and database integrity depend not only upon the KAPSE, but also upon the databases (composite objects) built with the KAPSE to store the data of specific projects. This motivates the differentiation between "distinguishing" and "non-distinguishing" attributes, the introduction of "capacities" as named abstract roles, and "private objects" as a mechanism for project-unique controls to be positively *enforced*.

This design may be characterized by envisioning the KAPSE as the active part of the database, rather than envisioning the database as the storage part of the KAPSE. This view, which will be evident in the following sections, considers the various active program contexts comprising the KAPSE to be elements of the database. The KAPSE guards the <u>entire</u> database, including those active components which implement the KAPSE. This was selected as the most promising approach to developing a design without security loopholes.

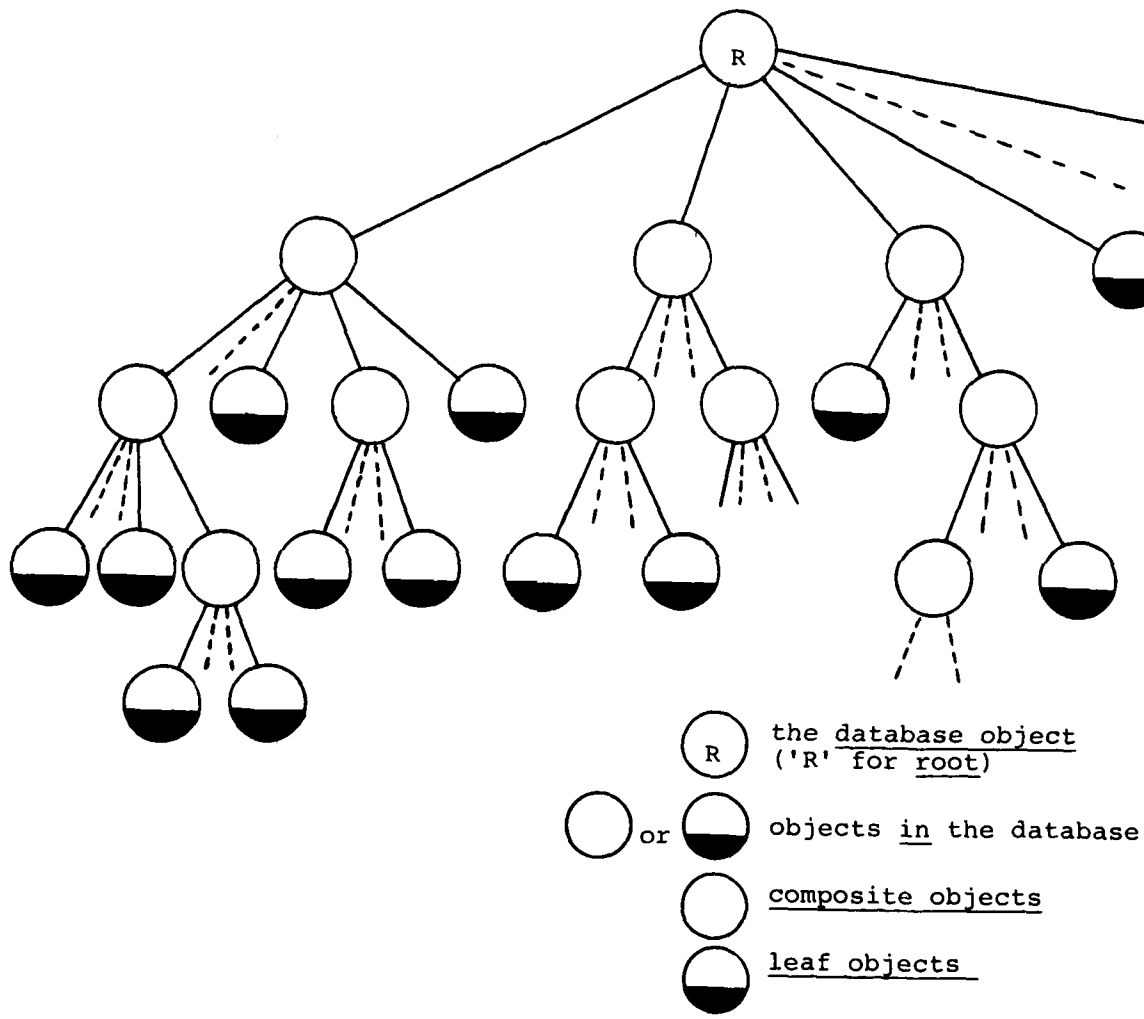
THE KAPSE AND THE DATABASE

Most KAPSE functions operate on the database, and the primary responsibility of the KAPSE is to protect the database.

The database includes the KAPSE specifications and implementations. The services provided by the KAPSE are the sole <u>connectors</u> between storage parts of the database, users, and active parts of the database, such as the KAPSE archive management and history keeping functions. For convenience of exposition, an outer, passive, view of the database structure is presented first, below.

DATABASE TREE STRUCTURE

The database is a composite object whose structure is defined in terms of sub-objects which may, in turn, be composed of sub-objects. Thus the database can be viewed as an object tree [STONEMAN, 4.B.1].



A sub-object of an object is called a component of the object. Leaf objects have no components.

Figure 4-1

KAPSE operations can modify the tree structure of the database -- by the creation of new objects and the deletion of old ones.

WHAT IS STORED IN THE DATABASE

a. Simple data objects -- documents, source code, text data, help files, etc.

b. Composite data objects -- Ada libraries, directories of tools, etc.

c. Executable objects -- programs, together with all of the context information which they require to run.

d. Executing objects -- the same as b), but in the state of execution. Executable, or executing objects are called program context objects.

e. Device objects -- objects which represent a physical device to the KAPSE. Every KAPSE function which serves or drives a device, such as a terminal operates on the corresponding device object.

f. User context objects -- a composite object which contains windows and other data associated with a user.

g. Window objects -- a special class of objects which defines the access paths through the database, as described later in this chapter.

LEAF OBJECTS

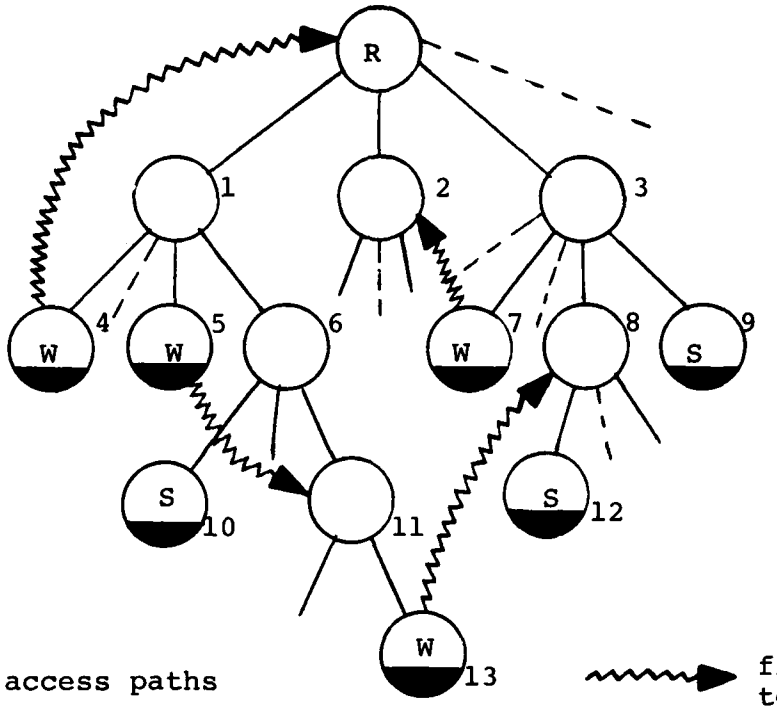
Leaf objects of the object tree are either windows or simple data objects. The principal characteristics of simple data objects are:

- the semantics of the internal structure is not relevant to KAPSE;
- can be manipulated by Package INPUT_OUTPUT;
- has a specific physical layout (used by Package I/O) -- direct access, indexed access, terminal I/O;
- is a unit of history keeping.

DATABASE ACCESS STRUCTURE

The access structure of the database at a given time is defined simply by its tree structure and window collection.

Each window provides access to all or part of a particular target object.



possible access paths

a. 4→R→2

b. 4→R→3→8

c. 13→8→12

d. 5→11→13

e. 5→11→13→8→12

f. 4→R→3→8→12

g. 13→8

Note: every access path begins with a window

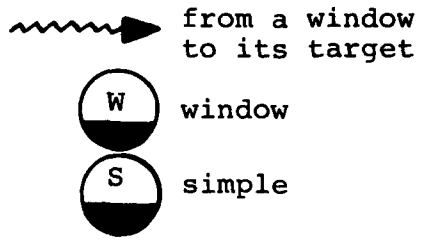
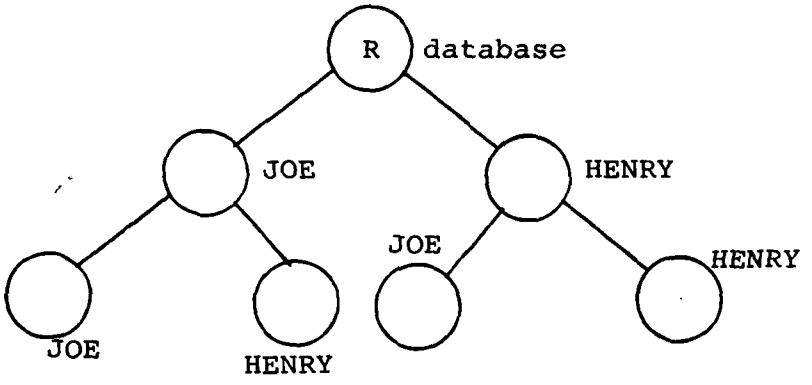


Figure 4-2

A program gains access to some designated object via an access path which begins with a window in the program context. Associated with each window is an access capacity, which determines the kind of access permitted within its target object. Therefore, two different access paths followed from the same window to the same end object will imply different restrictions on what operations can then be performed upon it.

OBJECT NAMES, OBJECT ADDRESSES

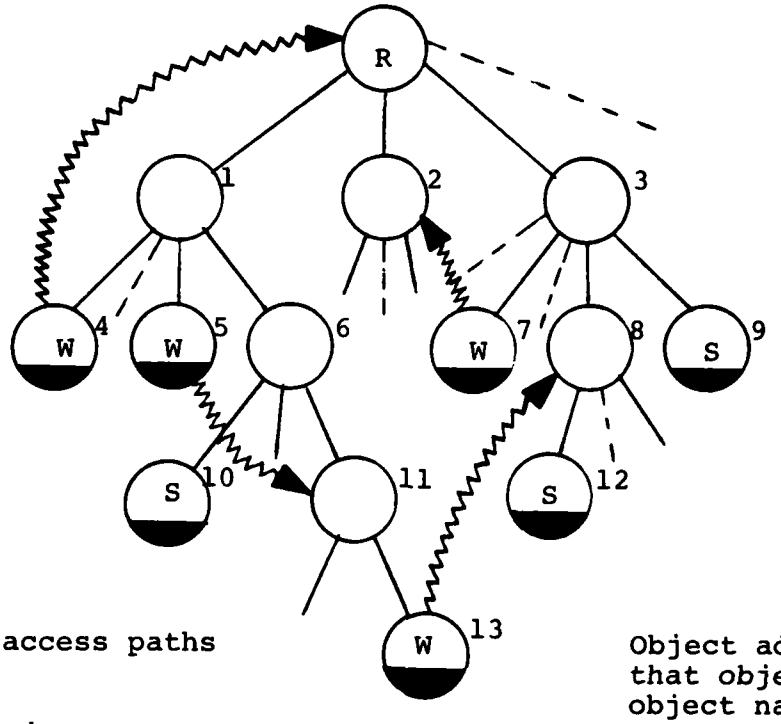
The name of every object is understood as relative to its enclosing composite object in the object tree, and distinguishes it from all other components of the same composite object. A complete unique name would start at the root of the database, and successively select the component which encloses the object of interest.



The name rule stated above is satisfied.

Figure 4-3

While every object has a unique name [STONEMAN,4.A.3], it can be addressed in many different ways. The description of an access path which ends with an object is an address of the object.



possible access paths		Object addresses (assuming that object numbers are object names)
a.	4→R→2	4.2
b.	4→R→3→8	4.3.8
c.	13→8→12	13.12
d.	5→11→13	5.13
e.	5→11→13→8→12	5.13.12
f.	4→R→3→8→12	4.3.8.12
g.	13→8	13.

Figure 4-4

Since windows point at their targets, the addresses do not need to specify the step from a window to its target. Therefore, the addresses shown on the right are generally shorter than the corresponding path descriptions on the left [STONEMAN, 4.A.4].

In lines c, e, and f, there are three different ways of addressing of object 12. Given a program operating in context 11, the KAPSE would locate object 12 in response to the address 13.12; a program running in context 1 could use either the

address 5.13.12 or 4.3.8.12 to reach object 12, but with two different sets of limitation as to what operations it could perform on object 12 after the object was found. In line g, the terminal dot in the address signifies that the target of the window is the object addressed.

DISTINGUISHING ATTRIBUTES AND NAMES OF OBJECTS

Every object is characterized by values for its attributes. In particular, its name consists exactly of the values of its distinguishing attributes, written in a labeled aggreate notation, or in a pre-determined order separated by dots [STONEMAN, 4.A.4 and 5.A.1].

All the components of a given composite object are characterized by exactly the same set of distinguishing attributes, and the values taken together represent a name used to select the component.

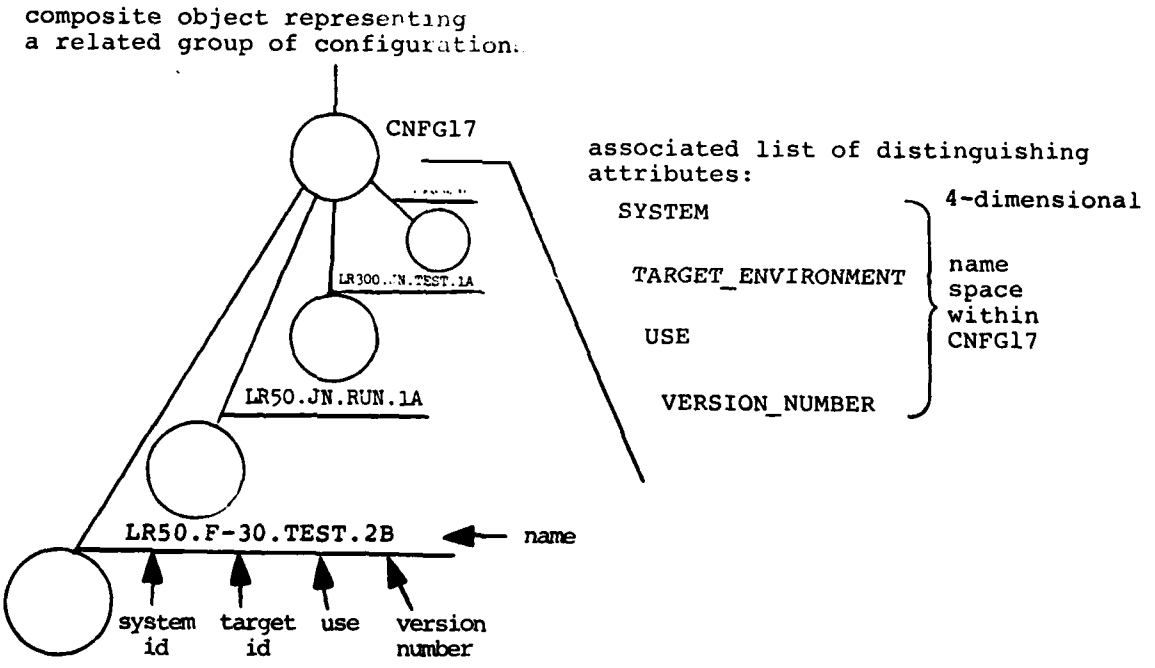


Figure 4-5

NONDISTINGUISHING OBJECT ATTRIBUTES

Every object must have some (at least one) distinguishing attribute; every object, may also have any number of nondistinguishing attributes. Which distinguishing attributes apply to an object depends on where it resides; the values of these attributes are part of the object's permanent identity. Nondistinguishing attributes are assigned to an object by users with appropriate rights [STONEMAN, 5.A.4 and 5.B.1]. Nondistinguishing attributes and their associated values can vary over the lifetime of the object, from its creation until its deletion.

SYSTEM ATTRIBUTES

Every object also has three system attributes -- attributes of special, system-wide relevance to KAPSE operations, as follows:

CATEGORY
ACCESS
HISTORY

The CATEGORY attribute protects the integrity of the object. It is used to describe those aspects of an object that must not be changed by any operation. Since there are operations which change the attribute values of an object as well as its content, the CATEGORY attribute can be used to guard against impermissible changes in both respects. Here are some critical examples of what CATEGORY can be used for [S.O.W., 4.1.2.9.2; STONEMAN, 5.A.6 and 5.B.2].

To guard against:

- Format or layout violations of simple objects;
- Creating an object component with an excluded name;
- Assigning excluded values to nondistinguishing attributes.

The ACCESS attribute indicates which executing programs (and therefore, indirectly, which users) can perform various operations on the object.

The HISTORY attribute gives access to the history of an object [STONEMAN, 4.B.4]. The type of history information maintained depends on the nature of the object. For a derived object, constructed as an output of a program applied to some set of input objects, the relevant history includes a script of the program invocation recording the state of the input objects, and the parameters provided.

For a source text object, created and edited by a human using a text editor, the history includes sufficient information to reconstruct the content of the object and the purpose for the editing session. The history does not record the actual keystrokes used to effect the editing.

The history for both types of objects are stored in a special portion of KAPSE memory which physically extends to off-line storage [STONEMAN, 4.A.6, 4.A.11, 5.A.5, and 5.A.8].

ACCESS CONTROL -- INTRODUCTION

Ultimately, all access control schemes are directed toward defining who is allowed to operate in what way on what objects. Every program execution is, ultimately, an execution on someone's behalf. What operations the program may perform must depend on what operations the person in question is allowed to perform. Such controls must apply to every kind of object: data objects, executable or executing objects, user context objects, device objects or windows [S.O.W., 4.1.2.10; STONEMAN, 4.B.6 and 5.A.7]. The operations under KAPSE access control divide into two classes: KAPSE operations and higher-level operations (as embodied in an executable object).

The KAPSE approach to achieving the required control objectives is accomplished with economical and flexible means. It should be noted that the problem to be solved is intrinsically complex. A single user's right/obligation to perform a certain function will, in general translate into many different permissions/limitations on KAPSE operations, depending on which object is considered. A reviewer, for instance, must be able to read the files he reviews and edit the files in which he records his results [STONEMAN, 4.A.9].

The access possibilities of a given programmer or manager change from time to time; in general, they are the result of several management decisions, and high-level synchronization requirements [STONEMAN, 4.A.9].

The current state of user access capabilities must, further, be translated into low-level synchronization requirements to insure low-level database consistency [STONEMAN, 4.A.12].

When a program wishes to operate on an object, the access controls that are applied depend solely on the access path implied by the address given. Further, at every step along that access path, the control to be exercised is subject to modification.

The access attribute of an object, combined with the capacity associated with windows in the database, determine dynamically the access rights available to an executing program. The value of the access attribute has the form of a table which lists

capacities and, associated with each entered capacity, a list of primitive access rights:

```
Example:
     simple object: SPEC10
         attribute: ACCESS:
                    value:
                                  capacity:    EDITOR

                    primitive access
                              rights:    read,
                                         write,
                                         append,
                                         modify user attributes
                                         read all attributes

                                  capacity:    SUPERVISOR

                    primitive access
                              rights:    read-delete
                                         copy
                                         modify all attributes
                                         read all attributes
```

Figure 4-6

A single capacity as specified in a window may translate to many different combinations of access rights, depending on which object is addressed through the window. This is as it should be. The editor, in his capacity as "editor", may need to read some objects, without being able to modify them; to append to other objects, without being able to modify them; to copy some objects, but not others, etc. The definition and designation of a capacity corresponds semantically to a higher level function, or role which, to be performed, requires various access rights to various objects.

Furthermore, since the actual powers which a capacity confers upon its holder depends on the ACCESS attribute of the objects affected, those powers may be changed without capacity revocation. For instance, suppose that the exercise of a capacity requires reference to a data table which is, at some point, discovered to contain a significant error. The actual power to read the table can be suspended by changing the access attribute of the table without searching out and revoking the capacities which require use of the table. In any case, these capacities may well permit useful work to continue -- at least for those operations not demanding the table. When the table has been repaired, its access attribute will be restored to make the table usable.

Now we note: Each window names not only a target object, but also a capacity.

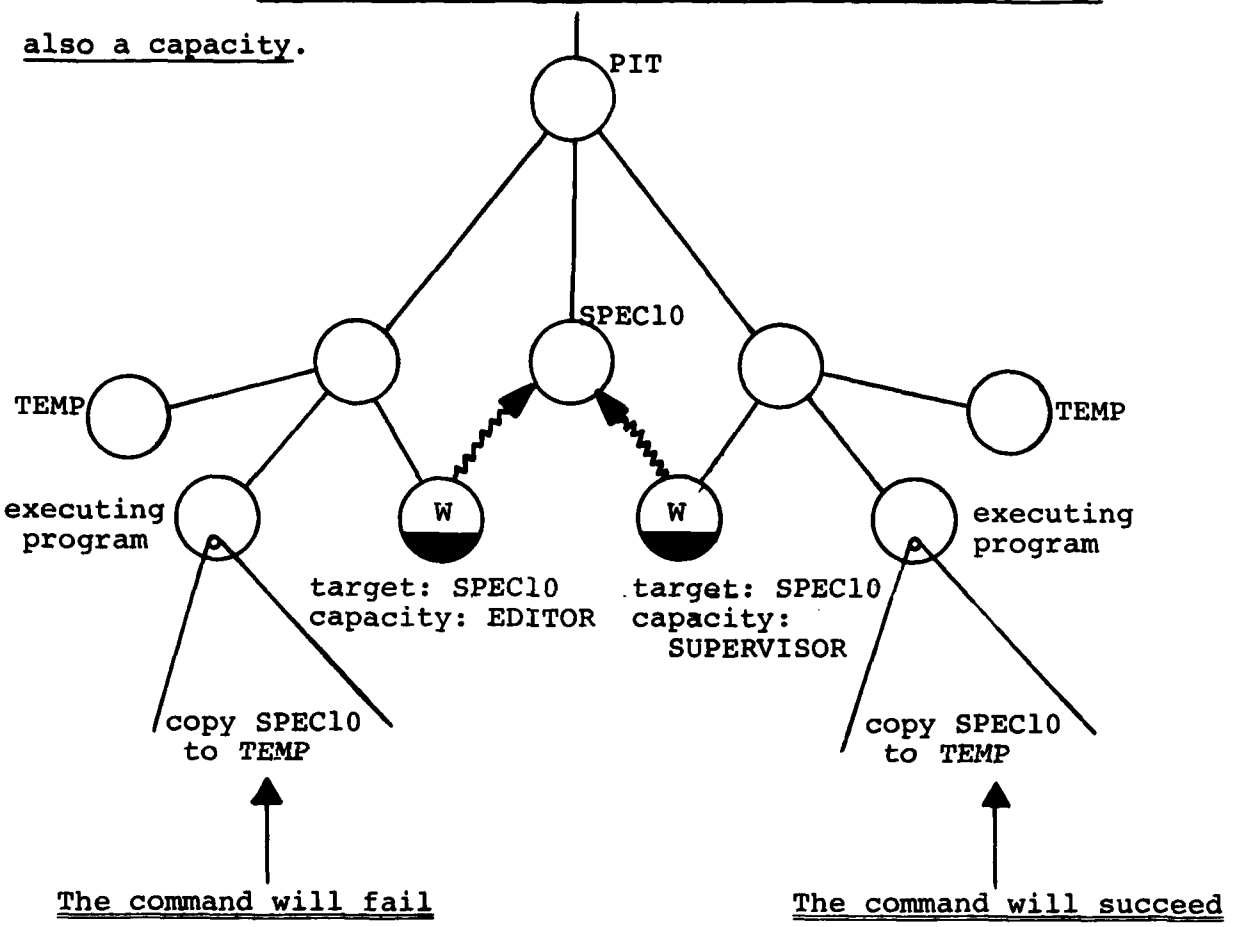


Figure 4-7

Most of the capacities which will be in use will be user defined. There are some few capacities which are system defined. These are discussed in the KAPSE B-spec.

A window on a composite object may allow the window holder to select components of the object -- for instance, for deletion or listing - or to select a name within the object with which to associate a newly created object. By means of an additional window part called a partition specification, the window can restrict an accessor's view to only a portion of the name space within the composite object [S.O.W., 4.1.2.7; STONEMAN, 4.A.7 and 4.B.5].

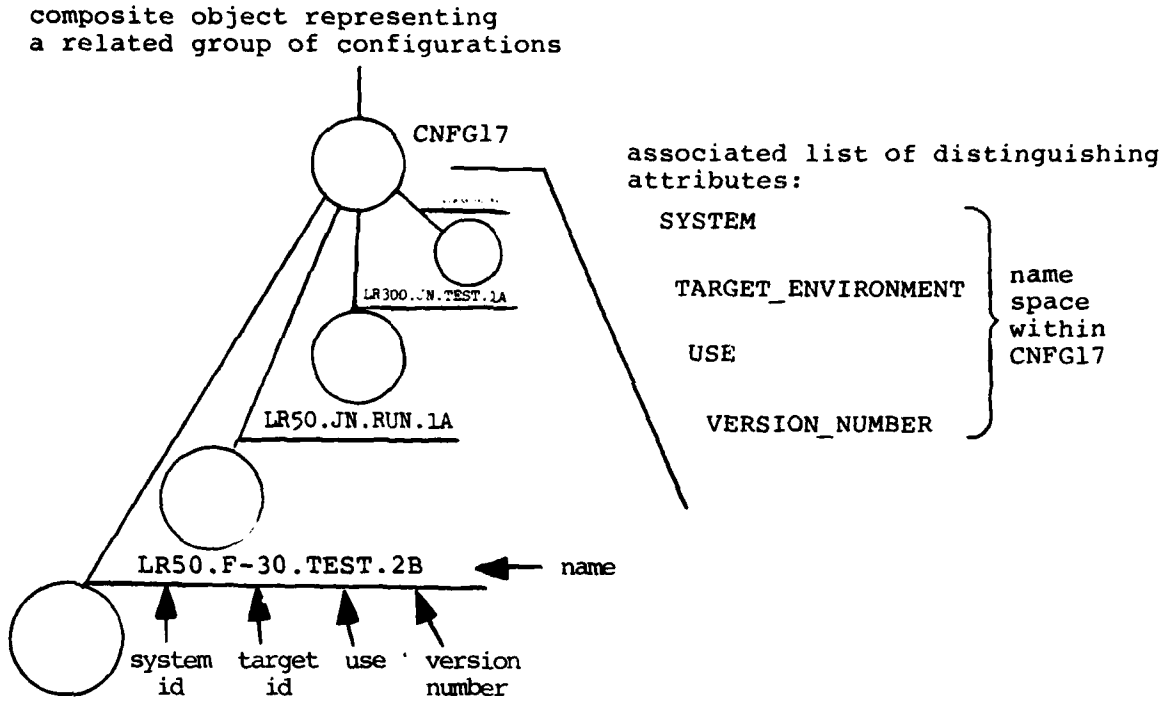


Figure 4-8

Examples of partitions:

(a)	**SYSTEM**	=> LR50	selects all names with "LR50" as first segment. Semantically all configurations of system LR50
(b)	**USE**	=> TEST	selects all tests configurations for target F-30 with version numbers 2A, 2B, or 2C
	TARGET	=> F-30	
	VERSION_NUMBER	=> (2A, 2B, 2C)	

The next figure illustrates:

- The effect of the partition specification in a window
- The effect on access rights resulting from an access path which goes through a composite object.

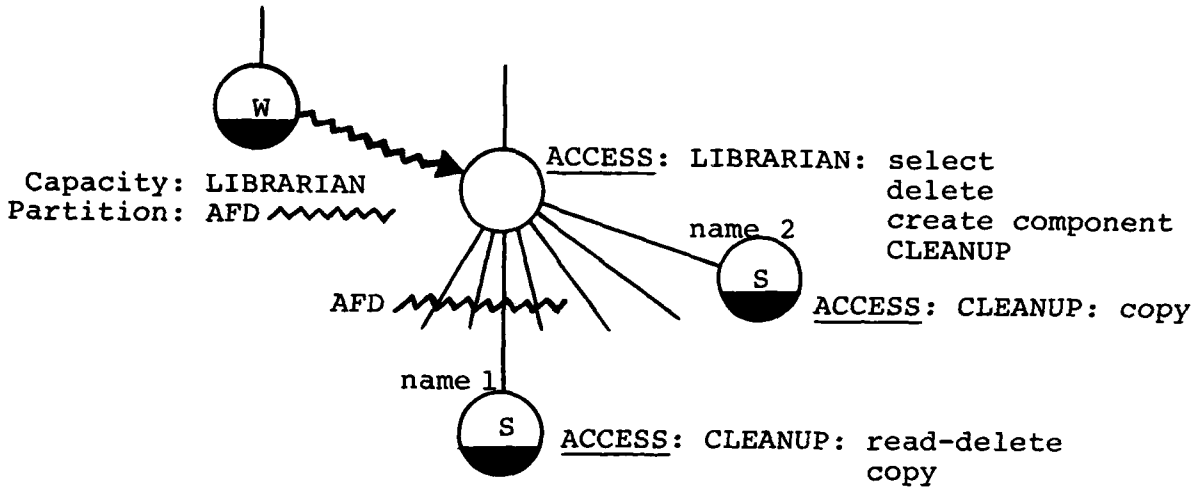


Figure 4-9

Suppose that a program attempts to execute the DELETE operation on the object with name 1. Suppose also that the access path implied by the address passes through the window shown in Figure 4-9. The DELETE operation would succeed because: (a) name 1 is shown as within the partition specified by the window; (b) the access attribute of the target of the window permits LIBRARIAN the primitive access right of component deletion. A similar attempt to delete the object with name 2 will fail because of the partition restriction.

Partitions play an important role in a number of KAPSE operations. For example, an operation called LIST_PARTITION lists the components of a partition by name. (This operation plays an important role in configuration management.) An attempt to list a partition as in Figure 4-9 will fail if the window used does not provide access to that partition.

Now suppose that figure 4-9 were modified by removing the window partition restriction. In this case, a program attempting to copy the object with name 2 via the window would succeed

because the access attribute of the window target allows the access rights of CLEANUP to LIBRARIAN. Relative to the window target, "CLEANUP" is called an internal capacity. Thus we see that in stepping past a composite object in following an access path, any external capacity rights are either translated into primitive access rights on that object, or translated into internal capacities for further translation.

An access path commonly passes through many windows. At each window, the access attribute of the window grants or denies the primitive access right 'go through' (see Figure 4-9 to the capacity obtained by previous steps of translation. If 'go through' is granted, then the capacity tested against the access attribute of the next object encountered is that of the window just gone through.

HIGH LEVEL ACCESS CONTROL

Suppose it is desired only to operate on an object -- call it HENRY -- by means of particular programs. HENRY, we shall suppose, is subject to certain high level consistency requirements, not expressible by means of its category attribute. HENRY's consistency can only be assured if its manipulation by users is restricted to specific programs. (It is this same problem which led to the invention of the Ada private type.).

This protection can be attained by creating a composite object -- call it HENRY' -- of which HENRY is a component. Program contexts for each of the special HENRY-manipulating programs are further components of HENRY'. The general user of HENRY is given a window on HENRY' which yields the "select component" right on HENRY', and the "program initiate" right on the program contexts in HENRY', and no other rights whatever.

Normally, when a user initiates a program using the Command Processor, a copy of the executable program context is created so that the same program may be running concurrently on behalf of more than one user. However, under the circumstances which led to a construct such as HENRY' above, it would be natural to guard against multiple concurrent executions of the special programs which operate on HENRY. To this end the KAPSE provides a special object type called a private object. The structure of a private object is exactly as described for HENRY' above. The special programs in the private object are called its operations. Each operation of the private object has an access attribute which determines which capacities may invoke it. Thus, different capacities may lead to different rights in regard to which operations can be invoked. The key difference between a private object HENRY' and an ordinary HENRY' as described above is this: When an operation of a private object is invoked, its program context is not copied into the user context before it is initiated. This inhibits the concurrent execution of the private object operations.

Access control mechanisms of the KAPSE are, in summary:

- The category attribute - used to assure the maintenance of relatively low-level consistency requirements of objects.

- The access attribute in association with windows - used to control what users can perform what operations on what objects.

- Partitions in association with windows - used to control the application of operations to designated classes of objects, instead of to single objects at a time.

- Private objects - used to assure the maintenance of relatively high level consistency requirements of objects.

In closing, we mention that there is an additional KAPSE mechanism concerned with synchronization -- and hence access control -- which we have not discussed above: the RESERVE mechanism. There are also additional refinements for the protection of attribute value settings. These latter mechanisms the reader will find described in the B-specification of the KAPSE.

PARTITIONS, COMPOSITE OBJECTS, AND MANAGEMENT

The structure of the KAPSE database lends itself naturally to a variety of project management and configuration control tasks, as illustrated in the following example. Imagine that the operational software for the F99 aircraft consists of four major packages:

EXEC - The real time operating system

NAV - The guidance and sensor management software

UI - The user interface: displays and controls

WEAPONS - The tracking, fire-control and defensive systems.

Within the software development facility, there are four "labor pools" or functional organizations: QA, Programming, Analysis, and Documentation.

Furthermore, four versions of the aircraft are to be built: versions for the Army, Navy, Air Force, and one for export. This

arrangement may be summarized as:

Module:	Exec, Nav, UI, Weapons
Department:	QA, Prog, Analysis, Doc
Version:	Army, Navy, AF, Export

These management dimensions correspond to the three dimensions of a composite database object created to contain the project data. The components have distinguishing attributes MODULE, DEPARTMENT and VERSION, with the possible values as listed above.

Using this arrangement, the project manager can create appropriate management visibility and control. The following actions would be typical.

Give the manager of each department an owner window on a two dimensional slice of the composite object. For example, the QA manager would be able to see all components within the partition "*.QA.*". His view of the database (through the window) is a 4X4 matrix of the Army, Navy, Air Force, and Export revisions of the Exec, Nav, UI, and Weapons modules. He can refer to objects by names such as "Exec.Navy", "Nav.Army", and "Weapons.AF". Each of these objects may be composite, containing, for example, simple objects TEST_PLAN, LATEST_SOURCE, BASELINE_CODE, STATUS_TEXT.

The QA manager may create windows of lesser capacity. He might, for instance, appoint a coordinator for the testing of each of the four systems (Army, AF, etc.). He would define a COORDINATOR capacity for these windows, which would include write access to the LATEST_SOURCE and BASELINE_CODE objects. Only the QA manager and his designated coordinators (managers with access to the coordinator windows) could accept a new release into the QA baseline. The QA manager would create another capacity, TESTER, and give the coordinators one dimensional windows of this capacity on, e.g., *.QA.NAVY,*.QA.AF. The Air Force QA coordinator would subdivide his tester window, giving a portion to each subordinate. He has considerable flexibility in this. He could give each tester the whole partition, "*.QA.AF", or could give one tester "NAV.QA.AF", one "EXEC.QA.AF", etc., or some combination with any desired overlap. The tester capacity, in any case, would have rights to execute BASELINE_CODE, to read TEST_PLAN and LATEST_SOURCE, and to both read and write STATUS_TEXT.

This example illustrates one primary aspect of the database design. Rather than imposing a uniform "solution" to the software management problem, the design lets managers manage.

In the example, the F99 software project manager has made a number of global management stragegy decisions.

- He has adopted a three-dimensional view of the project database. (The dimensions are department, module, and system.) This indicates that his management approach will require answering questions like "what is the QA status of the Air Force Navigation Version?" or "what modules of what systems are receiving the most attention from the programming department?"

- He has delegated the setting of QA procedures to the QA manager.

- He has created non-overlapping partitions for the QA, Programming, Documentation, and Analysis departments. This means that files must be explicitly moved (released) from one department to another, rather than being shared. For example, the programming department may have any number of simple objects within the NAV.PROG.AF component of the F99 composite object. These simple objects may contain several executable versions, in various states of debugging. None of these versions (revisions) are visible directly to the QA, Documentation, or Analysis groups. These groups may only test, document or analyze the revisions which have been specifically released by the programming department.

At the next level, considerable flexibility has been left to the individual department managers. The QA manager's options have been discussed above, at least by presenting a plausible scenario. In fact, he has the same degree of flexibility as the other three department managers. Decisions to be made at this level include:

- Whether to give access to the department's entire partition to all members of the department, or to subdivide the visibility.

- How to structure the object at each point within the department's partition. Within each object in "NAV.PROG.*", the programming department manager could create simple objects, "working", "tested", "integration", and "baseline" representing the various development states of each module.

- What capacity to use for the window assigned to each programmer. The programming manager might give developer windows with rights to the "working" and "tested" release levels to some programmers, and integrator windows with access to the "tested" and "integration" release levels to other programmers. He himself might retain the only rights to the "baseline" level, or might create coordinators for the Army, Navy,

Air Force, and Export systems with rights to the baseline release levels within those particular areas.

Depending on the size and complexity of the F99 system, and the number of people involved, managers at lower levels may have further decisions to make about sizes and capacities of windows for their subordinates, about the contents of components (the "baseline" object within the composite object "NAV.PROG.AF" might be a further composite object if the project is large enough), and about the access rights (such as read, write, execute) to be associated with specific capacities (such as tester, developer, and integrator) within his portion of the database.

The private object concept discussed earlier in this chapter allows the very easy development of tools for moving data between partitions, such as from "baseline" within NAV.PROG.AF to "baseline" within NAV.QA.AF. This "tool" would consist of the MCL "copy" command invoked by an appropriate guardian of the private object NAV.QA.AF.baseline.

The combination of windows, capacities, and rights with the "delegation" approach described in this section, and the special-case controls available through private objects, enable implementation of configuration control and management policies with no more "programming" than the writing of a few MCL commands. The approach is fully compatible with a spectrum of configuration control/managemenmt policies ranging from "just turn the programmer loose" to "access only by need-to-know" to "7-level incremental development and release strategy", or any combination of approaches at various points within the project organization.

KAPSE OPERATIONS

This section lists, with some discussion, the operations which, together with a number of type specifications, constitute the programmer's interface to the KAPSE.

It has already been mentioned that when KAPSE operations are performed on a user's behalf, it is some program which actually executes the request that the operation be performed. These requests for KAPSE operations appear in the requesting program as normal Ada procedure or function calls; with such a call, control passes to the KAPSE packages linked into the user's running program. Any Monitor service calls which need to be made in order to carry out the requested operations are then done by the KAPSE-package programming. Thus, the KAPSE provides an 'Ada Virtual Machine' interface to whatever hardware actually underlies the MAPSE.

The user's interest in the KAPSE operations lies in the fact that the Command Processor (Chapter 6) will be executing many of

them on his behalf, either directly, or by calling tools which will execute more intricate sequences of KAPSE operations to perform common functions. However, this list of operations is of most interest to the programmer, since they amount to the "operating system calls" of the Ada Virtual Machine for which he will be writing programs.

These operations are fully specified in the KAPSE B5 document, and hence we shall not repeat here the details of parameter types, required access rights, or failure conditions.

Operations Applying to All Objects

```
COPY (object, name)
DELETE (object)
RENAME (object, newname)
```

These three operations are likely to be called directly from the command processor, as well as from other Ada programs. At the time the COPY operation is performed, no additional storage is actually allocated, and no actual copying of bits is performed. Instead, "logical" copying is done: the newly-created 'newname' object contains a reference to the original 'object', and the reference-count at 'object' is incremented. Whenever any information in the object is actually changed -- via reference through whichever name -- just enough disk-blocks are physically copied (and references adjusted in the appropriate data-base objects) to maintain consistency of both "copies" of the object.

```
SET_ATTRIBUTE (object, attribute_label, attribute_value)
GET_ATTRIBUTE (object, attribute_label) --> attribute value
SET_ALL_ATTRIBUTES (object, attribute_values)
GET_ALL_ATTRIBUTES (object) --> attribute_values
PROTECT_ATTRIBUTE (object, attribute_label, yesorno)

SET_CATEGORY_ELEMENT (object, categ_elt, categ_elt_value)
GET_CATEGORY_ELEMENT (object, categ_elt) --> categ_elt_value
SET_CATEGORY (object, text_file)
GET_CATEGORY (object, text-file)
SET_CAPACITY_ACCESS (object, capacity, access_rts)
GET_CAPACITY_ACCESS(object, capacity) --> access_rts
GET_CAPACITIES (object) --> capacities

TRANSFER_BUDGETS (from_obj, to_obj, disk_amt, proc_amt)
```

These operations are the nuts and bolts with which the administrators and other users of the system put together the pattern of access rights desired for a particular project. In that use, the operations would normally be invoked by tools -- or Command Language scripts -- devised to assist the user in his

work.

Operations on Simple Objects

The following operations are applicable only to simple objects. A simple device object represents a host-machine physical device within the data base, so that the access control mechanisms can apply to attempted uses of the device.

```
CREATE (file_handle, object, file_mode)
CREATE_DEVICE_OBJ (object, device_name, root-window)
OPEN (file_handle, object, file_mode)
CLOSE (file_handle)
SET_FILE_INFO (file_handle, file_info_block)
GET_FILE_INFO (file_handle, file_info_block)
READ (file_handle, addr, size, num_rec, max_rec)
WRITE (file_handle, addr, size, num_rec)
```

In CREATE and OPEN, 'object' is the permanent name of the file object, while 'file_handle' is an Ada object which represents the file while it is open, and loses its meaning after the file is closed. CREATE also implicitly opens the new file.

Operations on Window Objects

The usage and properties of windows have been discussed extensively in the earlier part of this chapter. There are two operations concerned with the existence of a particular window:

```
CREATE_WINDOW (object, target, ancestor, partition, capacity)
REVOKE (super_window, sub_window)
```

And there are three operations, concerned with synchronizing access to regions of the data base, which require window names as arguments:

```
RESERVE (window, reserve_mode, time_limit)
RELEASE (window)
ABORT_RESERVE (window)
```

Operations on Composite Objects

The following operations (except CREATE_COMPOSITE) apply to all composite objects. Program-control objects and Private

objects have additional specialized operations, including their own CREATE_operations.

```
CREATE_COMPOSITE (object, attrib_labels)
OPEN_PARTITION (partition_handle, partition)
CLOSE_PARTITION (partition_handle)
GET_PARTITION_INFO (partition_handle, partition_info_block)
GET_NEXT_COMPONENT (partition_handle) --> component
LIST_PARTITION (partition, attributes)
```

In analogy to files, the composite object named in the 'partition' designator for OPEN_PARTITION is the permanent name of the composite object; the 'partition_handle' is a token for the (sub)set of components of that object to which access is requested. When the partition has been OPENed, GET_PARTITION_INFO provides the information necessary to step through the components in the partition with GET_NEXT_COMPONENT.

Note that the creation of an entire composite object is a rather more involved procedure than just calling CREATE_COMPOSITE: each component of the newly-created object must itself be created, with the appropriate CREATE_operation; and the component must be designated so that its name-string shows it to be an immediate descendant of this composite, with distinguishing attributes matching those given in the CREATE_COMPOSITE call.

LIST_PARTITION will in fact be a rather subtle tool -- it is the analogue of "DIRECTORY" commands in other operating systems -- displaying, in sorted order, all components in a designated partition which meet certain selection criteria.

Operations on Program-Context Objects

Except for the CREATE_operation below, these operations all deal with the activation of programs -- that is, they are run-time operation primitives, not data-base-structure modifying primitives.

```
CREATE_PROGRAM_CONTEXT (object, pure_part, impure_part)
PROGRAM_SEARCH (prog_name) --> prog_path
CALL_PROGRAM (prog_path, params, context) --> outparams
INITIATE_PROGRAM (prog_path, params, context,std_in,std_out)
AWAIT_PROGRAM (context, time_limit) --> outparams
SUSPEND_PROGRAM (context)
RESTART_PROGRAM (context)
PICK_PARAM (params, param_name, position, default) --> value
```

CALL_PROGRAM puts the designated program into operation then waits for it to terminate; INITIATE_PROGRAM puts the designated program into operation concurrently with the calling program. The 'context' argument to the operations is a local name for the particular instance of the invoked program, required only if subsequent operations will need to refer to it.

As with any composite object, the creation of a program context is an intricate process, but is usually done by the Linker tool -- in fact, the creation of a program context object is the operational output of the Linker.

```
IPC_ACCEPT (chan_nr, time_limit) --> call_block
IPC_END_RENDEZVOUS (call_block)
IPC_ENTRY_CALL (context, chan_nr, time_limit, call_block)
```

The above three operations allow concurrent programs to communicate with one another in a manner directly modeled on the tasking features of Ada; within an Ada program the tasks must agree on the format and meaning of the call_block, and on the channel numbers they will use.

Operations on Private Objects

Private objects are composite objects with particular attribute values, and the CREATE_PRIV_OBJ and ADD_OPERATION operations are simply procedures using the more elementary operations (see KAPSE B5 document); they are called out here as separate operations for completeness, and convenience to the programmer.

```
CREATE_PRIV_OBJ (object)
ADD_OPERATION (obj, opname, params, time_lim) --> outparams
```

User Service Operations

```
LOGIN (user_name, user_password)
LOGOUT
CHANGE_PASSWORD (password)
CHANGE_VIEW (partition)
CURRENT_USER_NAME() --> user-name
SEND_MAIL (user_name, subject, message, mail_seq_num)
SEND_MAIL_CHECK (mail_seq_num) --> yesorno
CHECK_MAIL() --> nr_new_mess
READ_MAIL (object)
```

When a user attaches his terminal to the MAPSE -- by means appropriate to the hardware system he is using -- he is prompted for his name and password, and a function represented here by the LOGIN "operation" is performed. When this operation succeeds, he will be in contact with the MAPSE Command Processor as the initial program executing on his behalf. The program-context object for the Command Processor will have accessible all the windows which define the rights of the user with respect to the objects in the data base.

While the user is working in a particular small area of data base name-space, he may call CHANGE_VIEW to temporarily restrict his "view" of the data base, to permit the use of shorter names, or to change the data-base environment of a program to be executed.

The other operations in this group provide a simple inter-user mail facility.

Packages

The remaining operations provided by the KAPSE can be grouped easily into functional packages, according to the Ada feature, or MAPSE tool, that they support. The names are mostly self-explanatory, and detailed descriptions are given in the KAPSE B5 document. These operations are simply listed here to provide a survey for the reader.

Debugging

SET_CURRENT_DEBUGGED_CONTEXT
GET_PROGRAM_STATE
CONTINUE
SET_PROGRAM_DATA
GET_PROGRAM_DATA
SET_ECP_BREAKPOINT
SET_EXCEPTION_BREAKPOINT
SET_TRAPS

Tasking

SET_DELAY
SIMPLE_ACCEPT
ENTRY_CALL
SET_OPEN
READY_TO_TERMINATE
SELECT_CALLER
ABORT_TASK
TERMINATE
CREATE_TASK
INITIATE_TASKS
RAISE_FAILURE

Storage Management

GET_STORAGE
FREE_STORAGE

Text I/O

SET_ECHO
NO_ECHO
SET_LINE_LENGTH
SET_COL
GET_LINE
SET_OUTPUT_INFO
GET_OUTPUT_INFO
SET_INPUT_INFO
GET_INPUT_INFO

Formatted I/O

CONV_FMT
FWRITE
FPUT
FEND
FREAD
FGET

Terminal I/O

READ_TERMINAL
WRITE_TERMINAL
SET_TERMINAL_INFO
GET_TERMINAL_INFO

KAPSE-Host Interface

ALLOCATE_BLOCK
INCREMENT_BLOCK_REF
DECREMENT_BLOCK_REF
READ_BLOCK
WRITE_BLOCK

Backup Recovery

FULL_BACKUP
INCREMENTAL_BACKUP
RECOVERY

History

GET_HISTORY_REF
RECREATE
NEW_SOURCE_ARCHIVE
OLD_SOURCE_ARCHIVE

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

```
NUM_REFS
GET_DERIVATIVES
SET_REFERENCE_LISTING
CHECK_REFERENCE_LISTING
GET_DIRECT_CONSTITUENTS
GET_SOURCE_CONSTITUENTS
GET_HISTORY_PARAMETERS
HISTORY_ACTIVATE
HISTORY_ON_LINE
HISTORY_TIME
HISTORY_MAKER
```

Chapter 5

PROGRAM COMPOSITION

Traditional programming systems have considered program composition to consist of two parts, the translation of source text into object modules by a compiler, and the linking of multiple object modules into an executable program. Ada has been designed with separate compilation features as an integral part of the language: packages may be separately compiled and made visible by means of with clauses; alternatively, procedures may be compiled separately and used by means of a body stub. In both cases, Ada requires language-defined consistency checking between separately compiled pieces. This requires that much of the integration work traditionally performed at link-time must, in the case of Ada, be performed at compile-time.

The design of the program library must satisfy the implied requirements set forth in the Language Reference Manual, must provide facilities for versions and revisions not directly supported in Ada, must serve as a communications vehicle between compiler and linker, and must be integrated in and utilize the KAPSE database system. Indeed, one major requirement on the database is that it in fact be an appropriate host for Ada program libraries.

Fundamental to the design of the program composition facility is the concept that the compiler, rather than the linker, integrates compilation units into the library. This approach satisfies the language-mandated requirements and also simplifies the design of related tools.

THE PROGRAM LIBRARY

The Program Library is implemented as a well-managed collection of database objects, whose contents are accessed via the Diana representation for Ada programs and the Virtual Memory Methodology system. A Program Library consists of all of the pieces of a set of related programs which are in various stages of development. The design maintains these compilation units in a well-defined state no matter how complex their interrelationships, or how drastically they change. Since the MAPSE supports multiple Program Libraries, parallel development of unrelated Ada programs may proceed without conflict or unwanted interaction.

Program Library Issues

The correct design and implementation of the Ada program library is a major key to achieving the goals of the MAPSE. Of particular importance are choices made in the representation of Ada programs and especially in the implementation of that representation in the light of separate compilation. These choices have a profound impact on the design of the MAPSE and its extension to an APSE. Those choices influence not only the internal design of APSE tools, but also shape the user's view of the system.

To satisfy MAPSE requirements, a program library must support

- efficient forms of representation of an Ada program during the process of compilation,
- permanent retention of and access to those forms which are needed by other tools, specifically the compiler when referencing separately compiled units or when recompiling a unit automatically, the Linker, the debugger, and yet to be conceived APSE tools, and
- addition (to those permanent forms) of as yet unspecified information produced by unspecified APSE tools.

Most importantly, separate compilation should introduce as few complications and as little overhead as possible to the fulfillment of these requirements.

A local view of the program library might, for example, be one in which the compiler produces a representation of each compilation unit that is completely self-contained. This would require the copying of information from the representation of other compilation units. An obvious drawback here is the space overhead for many copies of the representation of an imported type or specification, one in each referencing unit. With certain styles of programming-in-the-large, the space and time overhead of copying can be severe.

More serious is the impact this view has on the design of the MAPSE tools and the integrity of the program library. A tool which must look at the representation of more than one unit has to integrate the separate self-contained worlds. It must keep what has been defined by each unit, throw out what has been copied, and insure that the multiple copies are self-consistent. A simple example is the context specification WITH B,C where both B and C contain WITH A. Since B and C both contain copies of A the two versions of A must be sorted out. They might be different, in which case a choice must be made. This integration

must be performed each time a representation of a set of units is accessed. The integration exists only during program execution.

A global view, on the other hand, would be one in which all of the units are integrated together. The compiler produces a representation of a unit which is already integrated with the other units. Instead of copying information from a previous compilation, the information is directly referenced (pointed at). In a completely global view all traces of the original separation disappear after a unit is compiled, making it impossible to maintain multiple revisions or versions of a compilation unit. To fulfill APSE requirements, the global view must be adjusted to provide this capability.

The MAPSE program library is a logically integrated yet physically distributed world. Each compilation unit is represented separately in its own space (file), yet is integrated with the other units. This approach avoids the problems of both the local and the non-distributed global views. It avoids the overhead, complexity, and cost associated with copying. At the same time, it allows multiple versions of units to coexist in the same program library. The separateness of compilation units remains highly visible.

Program Library Architecture

A Program Library is a data base composite object, for example named MY_LIB, with four components names as follows:

```
MY_LIB.COMPILATION
MY_LIB.UNIT
MY_LIB.LINK
MY_LIB.LIBRARY
```

MY_LIB.COMPILATION contains a numbered component for each submission of text to the compiler, each perhaps containing multiple compilation units. Each such component has an attribute which preserves the options specified to the compiler; its content is the resulting abstract syntax tree with lexical attributes, a record of lexical and syntactic errors, and a representation of the source text for listing purposes.

MY_LIB.UNIT contains, for each compilation unit, a pair of components named MY_LIB.UNIT.DIANA.n and MY_LIB.UNIT.LIST.n, where "n" is a Virtual Memory Sub Domain (VMSD) number. The use of the VMSD number supports multiple revisions of the same compilation unit and is described in detail in the B-specification. Each component has attributes which identify, for example, the name of the unit, whether it is a specification or body, whether it is a package, procedure, function, or task, its

version name, its usage as regards recompilation, and its error status. The DIANA component contains the Diana representation of the compilation unit, integrating such traditional compiler outputs as object module, program tree, symbol table, and debugger table. The LIST component contains a printable listing for the compilation unit.

MY_LIB.LINK contains a version-named component for each executable program produced by the linker from compilation units in the library. One attribute's value is the name of the entry point while the second attribute's value identifies the program context composite object which contains the initial memory images as built by the linker, the memory map, and appropriate windows for execution.

MY_LIB.LIBRARY has attributes whose values are used for various program library housekeeping functions.

Figure 5-1 illustrates, in the style of Chapter 4, a Program Library as generated fresh for a user. Observe that it already contains compilations and compilation units corresponding to package STANDARD, the other Language Reference Manual defined package, the user-visible KAPSE packages, and the user-invisible run-time library package.

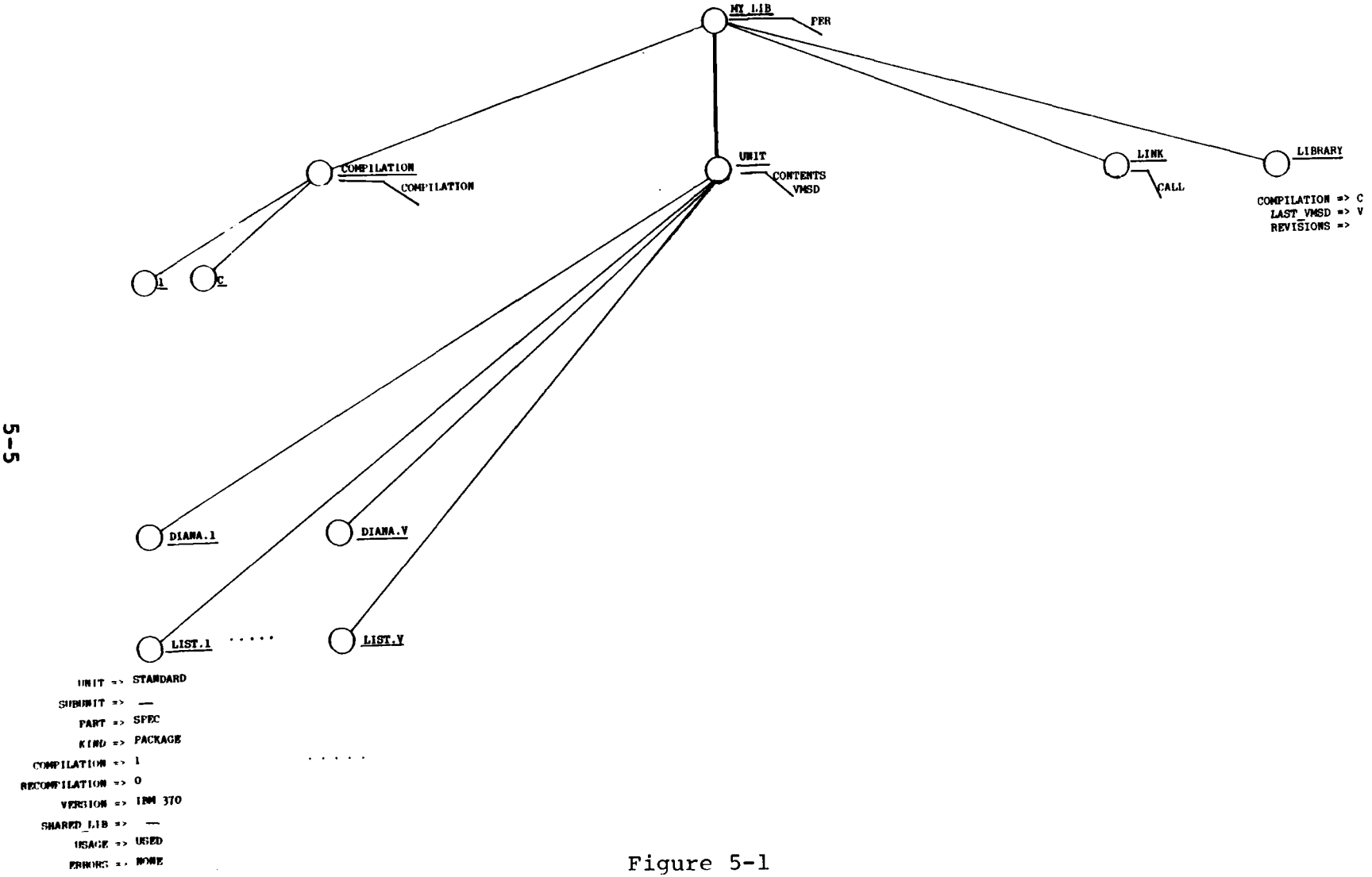

MY_LIB
PER
COMPILATION
COMPILATION
UNIT
CONTENTS
VMSD
LINK
CALL
LIBRARY
COMPILATION => C
LAST_VMSD => V
REVISIONS =>
1
2
DIANA.1
DIANA.V
LIST.1
LIST.V
UNIT => STANDARD
SUBUNIT => —
PART => SPEC
KIND => PACKAGE
COMPILATION => 1
RECOMPILATION => 0
VERSION => IBM 370
SHARED_LIB => —
USAGE => USED
ERRORS => NONE

Figure 5-1

PROGRAM COMPOSITION TOOLS

The major program composition tools are the compiler and the linker, although there are also a variety of support tools. These include:

- RECOMPILE - a program which allows additional control over recompilation.
- BODYGEN - a program which produces dummy bodies for those bodies which have not yet been written.
- PREAMBLE - a program which produces a preamble to enable an Ada program to be called by the Command Processor.
- CHANGE - a program which compares two Diana trees in order to determine whether recompilation is necessary.
- MAP - a program which stops the "domino effect" of unit recompilation.
- OBJECT_MODULE_CONVERT - a program which incorporates a "foreign" object module into the Program Library and allows it to be accessed via pragma INTERFACE.
- MAKE_LIBRARY - a program which creates a new Program Library.
- EXTRACT - a program which produces a text representation from the Program Library.

These support tools are described in more detail in the B-specification.

The major issues which the program composition tools must address are those of minimizing the recompilation and supporting multiple versions and revisions.

Recompilation Minimization

The source processed by the compiler may redefine a compilation unit which had been previously compiled and then referenced by another compilation unit, for example via the WITH statement. The redefinition implies that all referencing units must be recompiled to be affected by the change. Until the affected units are recompiled, there exists an inconsistency in the program library.

A program library can be well-defined and yet be inconsistent, as long as the inconsistency is properly

represented. For example, a specification might be changed without recompiling units which depend on the original. The units requiring recompilation are inconsistent until they are recompiled using the changed specification. The term "unit recompilation" refers to recompilation of a defined unit (usually to make it consistent) by effectively using the same source text as was submitted in a source compilation, but without requiring the user to actually resubmit a source text file. The abstract syntax tree, stored as a component of MY_LIB.COMPILATION, is used in recompilation; recompilation therefore bypasses the lexical and syntactic analysis performed by LEXSYN.

A single unit will undergo unit recompilation separately even though it was submitted together with other units in a source compilation. Unit recompilation will occur automatically according to the following two principles:

1) (Ease of use): The user never has to remember what or when to recompile. The system will always recompile whatever

2) (Efficiency): Automatic unit recompilation is delayed until USE of an inconsistent unit would otherwise result.

Both the compiler and the linker will perform automatic unit recompilation according to the above principles. These principles avoid wasted recompilations and allow the user to interact in various ways with the program library prior to the "domino effect". The domino effect refers to the widespread recompilations made necessary by a change to a low-level specification. The change must be propagated by recompiling all affected units (those which reference the specification), and then those which reference the recompiled units, and so on. By the second principle the unit recompilations do not occur when the low level specification is changed. Instead, they occur when the referencing unit is used as a referenced unit itself or by the linker.

An obvious example of the savings in recompilations occurs when the user is about to submit a different source version of an affected unit; prior unit recompilation would be a waste of time. A less obvious interaction with an even greater savings is the use of the MAP program. This program allows the user to assert that two versions or revisions of a compiled unit are identical. The program checks the assertion and if true, maps the old to the new. The mapping allows units compiled using the old version to be consistent with the new without recompilation, thus stopping the domino effect. The VMM system supports this mapping by translating automatically all of the old VM pointers to new VM pointers.

Versions and Revisions

The Ada language does not support the concept of versions for different purposes, or revisions in time order, of the same compilation unit. Such concepts clearly must be supported by the database; we have chosen to support them directly within the Program Library.

Each compilation unit has a VERSION attribute which serves to name the particular version. This attribute is set when the unit is compiled. In addition, the USE_VERSIONS option may be used to instruct the compiler as to which version of a particular unit should be used. If no version is specified, the unit with a null VERSION value is selected, implementing the concept of a "default version".

The concept of revision is implemented by means of the COMPILATION and RECOMPILATION attributes. The "most recent" revision is defined by the highest value of the COMPILATION attribute. If unit recompilation has occurred, multiple revisions may exist with the same COMPILATION value; in this case the one with the higher RECOMPILATION number is chosen. Only the most recent revision will have a USAGE attribute with a value other than OUT_OF_DATE.

EXAMPLE

Figure 5-1 illustrated the structure of the newly created Program Library MY_LIB. We shall now illustrate the results of compiling and linking two different versions of the same program.

Figure 5-2 is a single compilation unit, the example found in Section 10.1.2 of the Language Reference Manual. Let us assume it is stored in the database object named PROCESSOR_PROGRAM. The Command Language sequence

```
COMPILE     SOURCE => PROCESSOR_PROGRAM,
            LIB => MY_LIB

LINK        LIB => MY_LIB,
            MAIN => PROCESSOR,
            CALL => PROCESSOR_V1
```

will result in additions to MY_LIB illustrated in Figure 5-3:

- A C+1'st component of MY_LIB.COMPILATION containing the abstract syntax tree and related information,
- Components DIANA.V+1 and LIST.V+1 of MY_LIB.UNIT, with attributes as shown,

- A component MY_LIB.LINK.PROCESSOR_V1, which can now be directly executed.

```
-- in database object PROCESSOR_PROGRAM is

   procedure PROCESSOR is

     package D is
       LIMIT : constant := 1000;
       TABLE : array(1 .. LIMIT) of INTEGER;
       procedure RESTART;
     end D;

     package body D is
       procedure RESTART is
       begin
         for N in 1 .. LIMIT loop
           TABLE(N) := N;
         end loop;
       end;
     begin
       RESTART;
     end D;

     procedure Q(X : INTEGER) is
       use D;
     begin
       ...
       TABLE(X) := TABLE(X) + 1;
       ...
     end Q;

   begin
     ...
     D.RESTART; -- reinitializes TABLE
     ...

   end PROCESSOR;
```

Figure 5-2

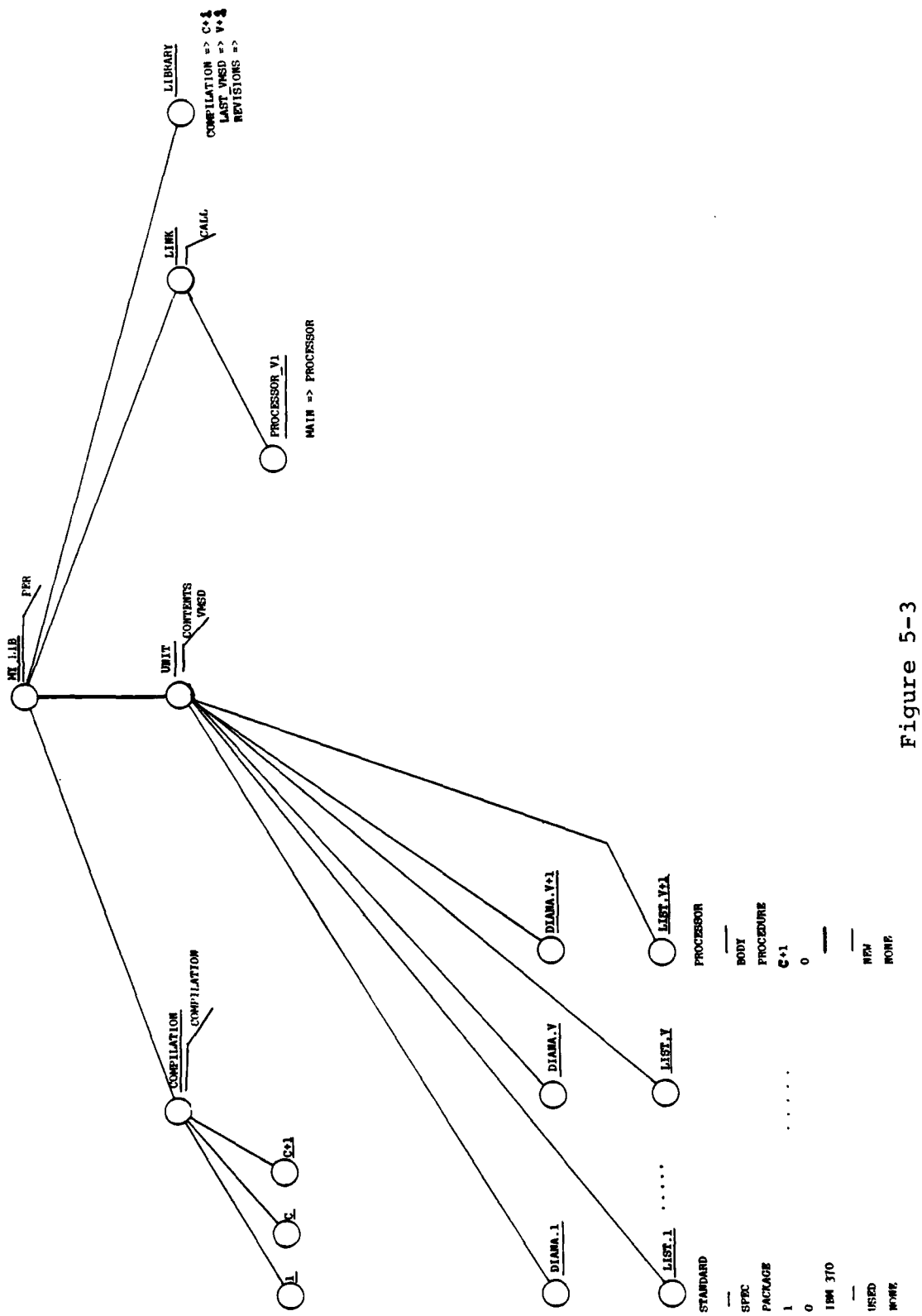

MY_LIB
PER
UNIT
CONTENTS
VMSD
LIBRARY
COMPILATION => C+1
LAST VMSD => V+1
REVISIONS =>
LINK
CALL
PROCESSOR_V1
MAIN => PROCESSOR
COMPILATION
COMPILATION
C+1
C
1
DIAMA.1
DIAMA.V
DIAMA.V+1
LIST.1
LIST.V
LIST.V+1
UNIT => STANDARD
SUBUNIT => —
PART => SPEC
KIND => PACKAGE
COMPILATION => 1
RECOMPILATION => 0
VERSION => IBM 370
SHARED_LIB => —
USAGE => USED
PRIVATE => NONE
PROCESSOR
—
BODY
PROCEDURE
C+1
0
—
—
NEW
NONE

Figure 5-3

```
-- in database object D is

   package D is
     LIMIT : constant := 1000;
     TABLE : array (1..LIMIT) of INTEGER;
     procedure RESTART;
   end D;


-- in database object D_BODY is

   package body D is
     procedure RESTART is
     begin
       for N in 1 .. LIMIT loop
         TABLE(N) := N;
       end loop;
     end;
   begin
     RESTART;
   end D;


-- in database object PROCESSOR_PROCEDURE is

   with D;
   procedure PROCESSOR is
     procedure Q (X:INTEGER) is separate;
   begin
     ...
     D.RESTART; -- reinitializes TABLE
     ...
   end PROCESSOR;


-- in database object Q is

   separate (PROCESSOR)
   procedure Q(X:INTEGER) is
     use D;
   begin
     ...
     TABLE(X) := TABLE(X) +1;
     ...
   end Q;
```

Figure 5-4

Figure 5-4 is a version of the same program, broken into four compilation units. If we compile and link them according to the scenario in Figure 5-5, the revised Program Library will be as in Figure 5-6. Note that the long form of the Command Language is used for ease of exposition; the short form is much more compact.

```
COMPILE         SOURCE => D,
                LIB => MY_LIB

COMPILE         SOURCE => D_BODY,
                LIB => MY_LIB

COMPILE         SOURCE => PROCESSOR_PROCEDURE,
                LIB => MY_LIB,
                VERSION => SEPARATE

COMPILE         SOURCE => Q,
                LIB => MY_LIB,
                USE_VERSION => SEPARATE

LINK            LIB => MY_LIB,
                MAIN => PROCESSOR,
                CALL => PROCESSOR_V2,
                USE_VERSION => SEPARATE
```

Figure 5-5

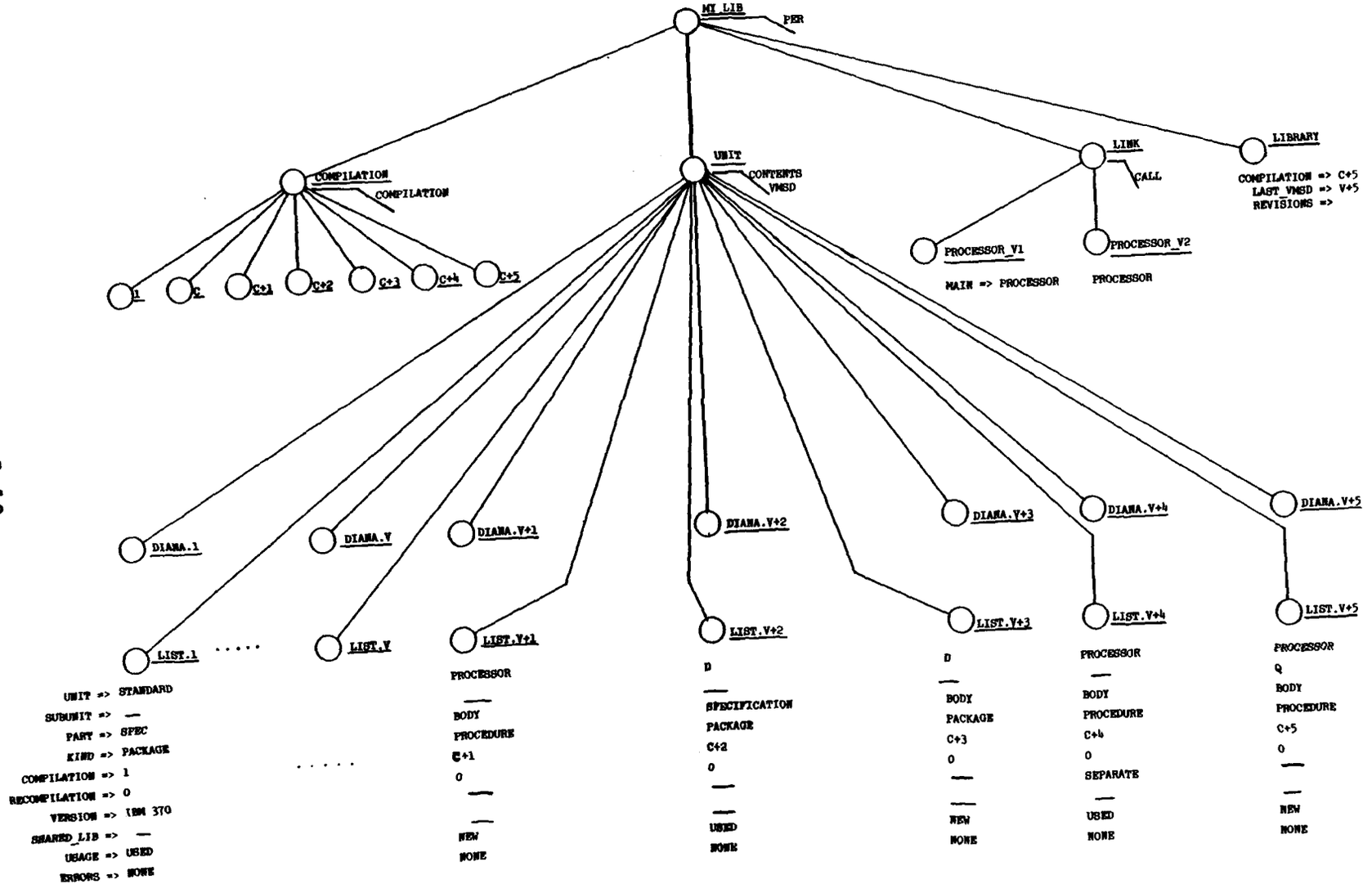

MY_LIB
PER
COMPILATION
COMPILATION
1
2
C+1
C+2
C+3
C+4
C+5
UNIT
CONTENTS
VMSD
LINK
CALL
LIBRARY
COMPILATION => C+5
LAST_VMSD => V+5
REVISIONS =>
PROCESSOR_V1
MAIN => PROCESSOR
PROCESSOR_V2
PROCESSOR
DIANA.1
DIANA.V
DIANA.V+1
DIANA.V+2
DIANA.V+3
DIANA.V+4
DIANA.V+5
LIST.1
LIST.V
LIST.V+1
LIST.V+2
LIST.V+3
LIST.V+4
LIST.V+5
UNIT => STANDARD
SUBUNIT => —
PART => SPEC
KIND => PACKAGE
COMPILATION => 1
RECOMPILATION => 0
VERSION => IBM 370
SHARED_LIB => —
USAGE => USED
ERRORS => NONE
PROCESSOR
—
BODY
PROCEDURE
C+1
0
—
—
NEW
NONE
D
—
SPECIFICATION
PACKAGE
C+2
0
—
—
USED
NONE
D
—
BODY
PACKAGE
C+3
0
—
—
NEW
NONE
PROCESSOR
—
BODY
PROCEDURE
C+4
0
SEPARATE
—
USED
NONE
PROCESSOR
Q
BODY
PROCEDURE
C+5
0
—
—
NEW
NONE

Figure 5-6

Chapter 6

COMMAND PROCESSING

The S.O.W. both recognizes the extreme importance of a well-engineered user interface and, at the same time, does not constrain the MAPSE designer to a particular style of interface:

> "...In general, the user will access the system to either invoke a MAPSE tool or to use KAPSE facilities to execute an Ada program (run a job) ... In the case where use of a tool is desired, the KAPSE must provide uniform facilities to "connect" the user to the appropriate tool. In the case of job execution, the KAPSE must provide standard interfaces to run-time support and host facilities required to perform that function...
>
> [S.O.W., 3.2.8]
>
> In order to execute jobs (user and tool), the KAPSE must provide job control facilities. This includes interfaces to required host facilities, a standard job control language, and functions to interpret and execute job commands..."
>
> [S.O.W., 3.2.8.2]

While recognizing the importance of "uniform facilities" and "standard...language" the S.O.W. requires neither a particular type of command language nor a particular implementation technology. The burden of designing an appropriate command language which is both natural to use and robust, and an implementation which provides both uniformity and flexibility, falls squarely on the MAPSE designers.

Our design of a MAPSE Command Language (MCL) and Command Processor (CP) is based on a number of assumptions:

- Most, but not all, users of the MAPSE and its extensions will use MCL. Some potential users, however, will require their own specialized command languages. It follows that the use of the CP by a user must not be built into the KAPSE, and assignment of a command processor must be settable at user and/or role definition time.

- The CP is a conventional MAPSE tool, coded in rehostable Ada. It follows that all facilities needed to implement the CP, including KAPSE and inter-program invocation and communication facilities, must be available to any Ada program.

- Although consistency requirements might argue in favor of Ada as a command language, Ada as it stands is not appropriate. However, where appropriate, MCL should be "Ada-like", and all facilities must be accessible both from the CL and from arbitrary Ada programs.

- Although the CP, Editor, Debugger, and future interactive tools have differing requirements, the overlap of requirements is considerable. All interactive tools need the ability, potentially, to be controlled by terminals or database objects, to update local environments, to process filed "scripts" of commands, etc. It follows that the design of the CP must be modular in order to provide the basis for future interactive tools.

MAPSE COMMAND LANGUAGE

Since the Command Language is the most visible interface a user has to the MAPSE, its design is critical. The MAPSE Command Language (MCL) must be intuitive and simple to use for conceptually straightforward applications and for casual users. At the *same time* it must be rich enough to support the most sophisticated user performing complex operations. This section discusses Command Language design issues, the elementary MCL features, and the advanced MCL facilities. Several examples are provided.

Command Language Issues

Users of newer programming and operating systems have come to expect an interactive command environment containing the facilities of a complete programming language, including features such as:

- Simple program execution and control

- Multiple concatenated program execution

- Execution control statements (if-then-else, loop)

- Command procedures and functions

- Command level variables

- Uniform access to tools and facilities

The MCL is an example of such an interactive command environment.

At the same time a casual MAPSE user must be able to perform productive work without the need of an MCL "refresher course". MCL has this property: a user may issue commands for service, including HELP, and receive immediate response. In addition, since MCL is a complete programming language, a user may compose arbitrarily complex command programs, for either foreground or background execution.

Command Language Style

There are three competing orientations for the design of a MAPSE Command Language:

- Ada-style
- English-style
- Functional-style

An Ada-style command language would utilize Ada, or a subset of Ada, directly. There are obvious advantages to this approach. Since Ada was designed for safety and readability, typical (though not all) MAPSE users will already be Ada programmers, and the command language processor would be able to share many tool components with the compiler. Ada, however, was designed for embedded computer systems, and its static syntax structure is not well-suited to the dynamic requirements of an interactive command system. Although an English-style command language with a flat, keyword syntax is tolerable for the occasional user, it is wordy and cumbersome for those who use the system on a day-to-day basis. A functional-style command language is typical in current interactive operating systems. Such a language allows the sophisticated user to take full advantage of a system's capabilities. It often, however, is difficult for a non-programmer to use, and, in any case, requires the user to learn a new language.

MCL represents a comfortable compromise among these three styles. Full Ada is not appropriate for a command language, but those portions of Ada that are appropriate have been used. This property allows carefully written sequences of MCL statements to be directly converted into Ada code. In the opposite direction, all system facilities may be invoked from within Ada code. Since MCL allows execution of an arbitrary Ada program, command sequences may be written in Ada, compiled, and executed from the terminal. In this particular case, Ada itself is available as a command language.

Use of a natural program invocation syntax, and the representation of all system facilities as programs, allows a near English-like simple command interface. For example, to compile an Ada procedure named TEST, one can type

```
COMPILE TEST MY_LIB
```

More sophisticated operations can be performed through the use of functional-type notation. For example, to compile TEST in the background, piping the compiler messages to a program COMP_STAT, a filter which accumulates compiler statistics in a data base object, and then to a line printer queue called LPT, a sophisticated user might type

```
COMPILE TEST MY_LIB -| COMP_STAT -> LPT -&
```

If this type of command were used often, the user might write a script, or an Ada program to carry out the function. If the program were named COMPILE, with two parameters named SOURCE and LIB, then

```
COMPILE SOURCE=>TEST LIB=>MY_LIB
```

or, more simply

```
COMPILE TEST MY_LIB
```

would carry out the new, user-specified, compile function. Observe that the new compilation user program effectively replaces the old predefined compilation tool for this user.

Command Language Requirements

Command languages deal primarily with names of programs, command options, and database objects; only more advanced users make explicit use of variables and control structures. This observation implies that program, command option, and database object names must be easy to write and manipulate in MCL.

Names of programs and database objects should not be bound to particular program and object instances until the time of actual use. Representation of names as strings meets this requirement; quotation of strings, however, results in an awkward command language. MCL allows strings which have the lexical form of a name to appear without quotes. The association of the name with a particular program or object does not occur until the name is interpreted by the CP. Thus, for example, the name of a program, being a string, may be passed as an argument to another program.

Command options behave much like enumeration literals. Unlike the literals of an enumeration type, however, the set of command options must be capable of extension. The treatment of command options as names, in the above sense, solves this problem.

Unfortunately, introducing such a simple quotation convention raises an ambiguity with command language variables. The set of names used for database objects ought t be disjoint from

the set of variable names. Since command variables are used infrequently, compared to database object names and program options, command variables must bear the syntactic burden of unique identification; a command variable name is an identifier prefixed with "%".

The fundamental purpose of MCL is to facilitate the invocation of programs. The command set has been chosen using a subset of Ada where relevant as well as several non-Ada notations.

Many features of Ada are appropriate for use in the command language. These are listed in Figure 7-2.

statements	examples
assignment	`%I := %I+1;`

if	`if %I<%J then`
	`  %K := %K+%J;`
	`end if;`

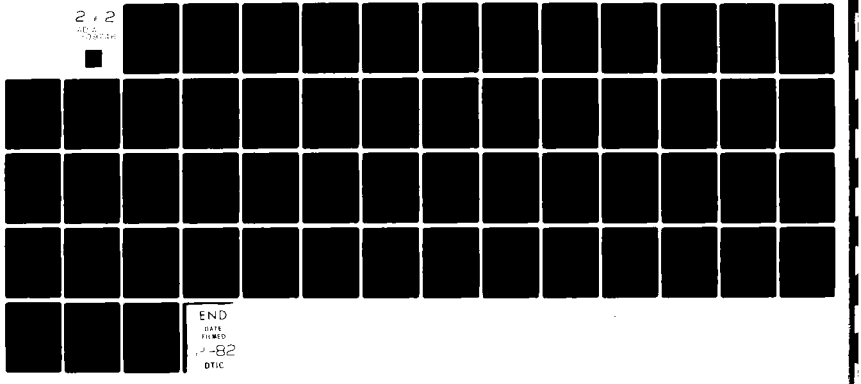
loop	`for %I in %J..%K loop`
	`  %S := %S+%I;`
	`end loop;`

return	`return;`
call	`COMPILE (SOURCE=>TEST);`
elements	
types	`INTEGER, BOOLEAN, STRING, REAL`
expression	`(%I = 12) and (%J >= %K)`
quoted string	`"ABCDE"`
decimal number	`12.23`
integer	`723`
identifier	`ABC`
attribute reference	`TEST'CATEGORY`
function call	`F(2,3)`
parameter mode	`IN OUT`
dot notation	`.CURRENT_DATA.TEST`

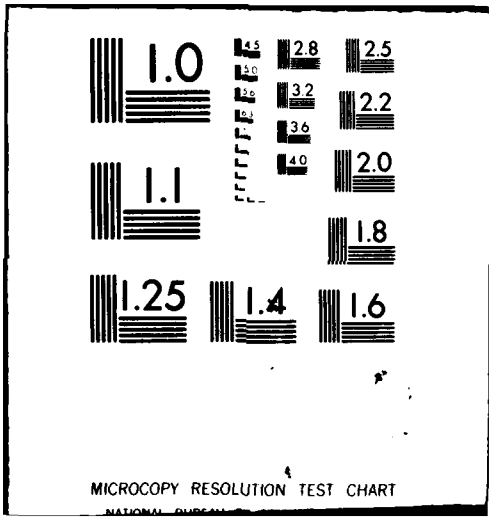
FIGURE 7-2: Ada Features Used in Command Language

Although usable as described, many of the Ada features are relaxed for ease of interactive use. For example, a statement may be terminated by end-of-line rather than ";", the parameters in a call need not be surrounded by parentheses, and the currently visible database object .CURRENT_DATA.TEST may be simply accessed as TEST. In examples, the Ada-like syntax is used in scripts and other retained command sequences, the relaxed syntax is used for user-entered commands; there are no semantic differences between equivalent syntactic forms.

AD-A109 746 INTERMETRICS INC CAMBRIDGE MA F/G 9/2
ADA INTEGRATED ENVIRONMENT I. DESIGN RATIONALE.(U)
DEC 81 F30602-80-C-0291
UNCLASSIFIED RADC-TR-81-357 NL



END
DATE FILMED
-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART

Other Ada facilities such as generics, full Ada typing, full tasking, and separate compilation are not relevant for a command language. On the other hand, a command language should have built-in facilities for such activities as input/output redirection and background execution, available as commands, and with appropriate operator-like syntax.

ELEMENTARY COMMAND LANGUAGE FACILITIES

When a user logs in, is identified to the MAPSE, and assumes a particular role, a role-specific program is executed. Typically, this will be an instance of the Command Processor invoked with a role-specific window into the database and a role-specific set of capacities, identifying, among other things, programs which may be executed.

Program Invocation

At this point a user might, by typing

```
HELP
```

inquire about the programs currently available. For more detailed information concerning a particular program, a user might type, for example

```
HELP  COMPILE
```

The user has invoked, via MCL, a particular program named "HELP".

HELP may also be obtained for a particular parameter of a program. This help may be requested in the course of parameter specification. For example:

```
COMPILE MYFILE ?
```

would cause the CP to print help information on the second parameter of COMPILE, then allow the user to specify a value for that parameter.

More generally, all programs are invoked via the same uniform access mechanism, regardless of whether they are implemented as

- a KAPSE procedure,
- a MAPSE program,
- a user-written program,

- a pre-defined tool, or

- a command language script.

If a CP command calls for a program to be invoked, all the windows named in the PROGRAM_SEARCH_LIST attribute associated with the CP's context object are searched for that program. This enables the user to specify program names using a convenient shorthand notation. The program call syntax is like the Ada procedure call syntax, with modifications to make it more convenient for interactive use. Like Ada, tool access allows both positional and named parameters, and IN, OUT, and IN OUT modes. Parameters may have initial values. Unlike Ada, positional and named parameters may be freely mixed. The CP parses the parameter association by grouping together all positional parameters, followed by all named parameters. For named parameters, the CP convention is that only the last occurrence of a parameter association is used, permitting a user to change a mistyped named parameter without retyping others. As in Ada, OUT parameters allow a tool to return values. Thus, a single program invocation may return multiple values in distinct CP variables which can in turn be used as IN parameters to other programs. Unlike Ada, the command language allows missing and default OUT parameters. In these cases, the CP generates an implicit CP variable declaration. The generated variable's name is the catenation of '%' and the formal parameter name. This variable is assigned the actual, or default, output value. If this generated variable conflicts with an already existing variable, the user is informed, and may specify that the variable's value should not be modified. Thus, the user may utilize OUT parameters as desired, without facing any added typing burden, and without the fear that default OUT parameters may wipe out useful data.

As an example, the following might be an Ada subprogram specification for a COMPILE tool:

```
procedure COMPILE
        ( SOURCE:  IN STRING;
          MSGS   :  IN STRING;
          ERRORS:  OUT INTEGER; );
```

With this specification,

```
COMPILE  TEST
```

will cause compilation of the current database object TEST, with messages being sent to the CURRENT_OUTPUT device. Since ERRORS, an OUT parameter, is missing, a variable %ERRORS will be installed with a value equal to the number of errors detected.

A number of tools come predefined in a typical user's window. These include:

EDIT	- invoke text editor
COMPILE	- invoke Ada compiler
LINK	- make an Ada library ready for execution
DEBUG	- invoke a program under debug control
HELP	- ask for help
GET	- set variables from a file
PUT	- write an expression
LOGOUT	- end the session

Names

The relative window through which the CP views the database is termed its local context object. All objects viewed through the local context object have names beginning with '.'. One such object in particular is .CURRENT_DATA. When a program is invoked, a new local context object is created, and a copy of .CURRENT_DATA is placed in it. Thus in a nested program invocation, all programs see the same objects in .CURRENT_DATA, and have their own local objects elsewhere in the local context object.

The command language naming convention assumes that any name without a prefix dot, such as GLOBAL, is in fact a reference to .CURRENT_DATA.GLOBAL and hence is visible to all tools. Any name with a prefix dot, such as .LOCAL, is in fact local to that invocation. Of course, .CURRENT_DATA.GLOBAL is just the full name for GLOBAL.

Attributes of database objects are named using the conventional Ada attribute notation. Thus D'HELP is the name of D's HELP object, and X'EXISTS is TRUE only if the object with name X currently exists.

Types

The CP does not allow the definition of arbitrary types. Rather a set of types is chosen which is most useful in interprogram communications: STRING, BOOLEAN, INTEGER, REAL.

Variables

CP variables are useful for storing values local to the CP. The interactive command environment argues against statically declared variables of fixed types. Rather, a CP variable is implicitly declared by its first use, and is always of type STRING (with implicit conversion to other types in expressions). This makes MCL smaller and easier to implement without any loss in functinality, since any value may be encoded as a string.

In combination with database objects, CP variables provide a range of possibilities for storing data, as follows:

- CP variables are local to the CP in scope, and are temporary in nature (i.e., disappear at the conclusion of the CP session).
- Database objects in the CP's context object are global in scope, but are temporary.
- Objects in any other window are global in scope and have a permanent lifetime (unless explicitly deleted).

Statements

The if-statement and loop-statement syntax and semantics are borrowed directly from Ada. The return-statement causes termination of the CP while a logoff-statement is an invocation of the KAPSE session protocol logoff mechanism.

The assignment statement allows the values of both variables and (assignable) database object attributes to be changed. For example

```
ECS_SYSTEM'RELEASE := ECS_SYSTEM'RELEASE+1;
```

The right-hand side of an assignment must be a string. If it is not, an implicit conversion is performed. GET and PUT enable the user to read values from standard input or write values to standard output. An expression, written alone as a statement, has the effect of PUT (expression); in other words, the value of the expression is sent to the current output device. Thus one need only type

```
TEST'HIGHEST_REVISION
```

in order to find the highest revision number for TEST.

ADVANCED COMMAND LANGUAGE FACILITIES

In addition to the MCL facilities thus far introduced, several facilities are available for more sophisticated users, including scripts, input/output redirection, piping, background tasks, blocks and execution of data as commands. These facilities provide convenient syntax for frequently used command language capabilities.

Scripts

A script is a sequence of CP commands stored in a database object. It is functionally equivalent to a linked executable program in terms of invocation syntax, help information and parameter passing. A script may receive IN parameter values and return updated OUT parameter values if it contains a subprogram simulation command. This command is similar to an Ada subprogram body. Indeed, scripts may be written which can, in fact, be compiled. The choice to compile or interpret is merely one of efficiency.

Examples:

Following is a script which implements HELP:

```
procedure HELP (%TOOL: IN STRING := ".CURRENT_DATA") is
begin
  if not %TOOL'EXISTS then
      PUT ("NO SUCH PROGRAM");
  elsif not %TOOL'HELP'EXISTS then
      PUT ("NO HELP AVAILABLE");
  else
      PUT ( CONTENTS(%TOOL'HELP));
  end if;
end HELP;
```

Associated with each database object is an attribute HELP which names an associated object. This object, if it exists, contains text help information which is printed. The CONTENTS function fetches the contents of the database object named in its argument.

The following script allocates a temporary library, compiles an Ada procedure into it, links the library, and runs the program under control of the debugger. Input to the debugger comes from the user at the terminal. When the user enters "return" to the debugger, it returns to the script. Termination of the script causes the library automatically to be deleted.

```
procedure ADACLG (%PROG:  in STRING)  is
begin
  MAKE_LIBRARY (.LIB);
  COMPILE (SOURCE=>%PROG
           LIB =>.LIB)
  LINK (.LIB);
  DEBUG (LIBRARY=>.LIB);
end ADACLG;
```

Input/Output Redirection

Programs which are invoked by means of the Command Language typically are written to accept input from the default input object (usually the keyboard) and provide output to the default output object (usually the display).

> "For all GET and PUT procedures, there are forms with and without a file specified. ...If no file is specified, a default input file or a default output file is used. At the beginning of program execution, the default input and output files are the so-called standard input file and standard output file, which are open and associated with two implementation-defined external files."
>
> [LRM, 14.3]

When a program is invoked, the Command Processor's default input and default output objects become the standard input and standard output objects for the program. The Command Processor's default input and output objects may be changed, utilizing facilities similar to those of Ada. For example,

```
SET_INPUT   BATCHED_INPUT
```

Commands will now be accepted from the database object BATCHED_INPUT until either a subsequent SET_INPUT command is accepted from *BATCHED_INPUT or BATCHED_INPUT reaches an end-of-file* state (in which case default input reverts to the CP's standard input).

It may be desired to locally change the default input and default output for one command. For example, ES may contain a standard editing script. If a user wishes to edit the program TEST according to the script, ignoring the editor's output, the command

```
EDIT  TEST -< ES ->  .NULL
```

will carry out that function.

Piping

More generally, a user may wish to execute a command, directing its output to a temporary database object, and then execute a second command on the results of the first command. For example, a preprocessor PRE might perform some actions on an Ada text. In order to preprocess an Ada text file named TEST, and then use it as input to a program FLIGHTSIM, a user might write

```
PRE SOURCE=>TEST, RESULT=>.TEMP
FLIGHTSIM  -< .TEMP
```

This can be written more conveniently as

```
PRE SOURCE=>TEST -| FLIGHTSIM
```

Pipes represent input/output connections between programs, files, or devices. They provide programs with a uniform input/output interface so that, once written, they can be connected arbitrarily to other programs, files, or devices. Utilizing pipes, complex applications can be developed by interconnecting smaller, simpler programs.

In order to use piping effectively, programs must be designed to utilize the standard text input/output facilities as an alternative to a specific IN parameter specifying input or output. In the previous example, FLIGHTSIM's specification might be

```
procedure FLIGHTSIM (SOURCE: STRING;
                     OUTPUT: STRING);
```

If specific input and output files are not provided, FLIGHTSIM assumes that input is to come from standard input and output is to go to standard output. Since piping connects the STANDARD_OUTPUT of one tool to the STANDARD_INPUT of another, this default usage allows for piped connections.

When temporary files are used, the temporary file must be large enough to hold all of the intermediate results. Piping establishes two tools as co-routines, with a finite amount of buffering.

As can be seen, MCL provides a variety of means of composing programs and passing information among them, including:

- standard I/O redirection
- pipes
- database objects, and
- parameters

Background Tasks

In general, a user of the Command Language will wish to have the ability to execute a command, or command sequence, in parallel with the execution of additional commands. For example, a

user may wish to run a lengthy compilation in parallel with an editing session. To accomplish this, a user might invoke a background task, as follows:

```
:COMPILE TEST -&
 TY executing
:EDIT OTHER_TEST
    ...
:STATUS TY
```

A background command is assigned a task name which may be used to control its execution. This name may be assigned by the CP, or may be specified by the user (by preceding the command with a label which serves as its name). The status-command will list the status of a background task while the WAIT command enables the user to wait for a background task's completion.

Full Ada tasking is too powerful a facility to utilize in a Command Language; thus a simpler, entry-less tasking model was chosen.

Jobs

A command's execution may involve the invocation of a program. Each program invocation, referred to as a job, has associated with it a context object which describes its execution. The context object name (which is formed from the invoked program's name) may be referenced to control the program's execution. Each background command maintains its own composite object of context objects for any programs it invokes. This enables the user to control all jobs within a background command in a convenient fashion.

Statements available to control a job's execution include:

- status: determine the program's status,
- stop: stop the program's execution (so that it may be debugged, e.g.),
- start: continue the execution of a previously stopped program, and
- cancel a program's execution, such that it cannot be restarted.

Blocks

It occasionally is desirable to group a number of statements together, for example, for execution as a background task or I/O redirection. The following block performs the execution of the preprocessor and flight simulator as a background task.

```
BEGIN
  PRE SOURCE=>TEST, RESULT=>.T
  FLIGHTSIM SOURCE=>.T
END -> ERROR_FILE1 -&
```

All error messages will be placed in the database object named ERROR_FILE1.

A static, compiled language such as Ada maintains a strict separation between programs and data: a program is a linked executable object, while that program's data may be of any format (text, Diana, e.g.). In a dynamic, interpretive environment such as MCL, in which both commands and data are text, this separation is irrelevant. The EXEC command enables the execution of data as commands. For example:

```
%A := COMPILE
EXEC %A& "MYFILE"
```

This facility enables the user to build commands using CP variables, strings and concatenation operators.

Example

The MAPSE Command Language does not provide a facility for "remembering" default parameter values between invocations of commands. For example, one might wish to support the concept of editing a (one procedure) program, followed by the ability to compile that program, without having to specifically name it. With such a facility a user could type

```
EDIT MY_PROG
```

followed by

```
COMPILE
```

without having to repeat the name MY_PROG. An approach to implementing this facility involves defining a "parameter object" in the current window, and writing scripts which take default values from the parameter object. Assuming the existence of a database object PARAM_OBJ, the following scripts accomplish this:

```
--Edit script contained in a data object named "EDIT"
  procedure EDIT (%PROG: in STRING) is
  begin
    PUT  %PROG ->PARAM_OBJ;
    .CURRENT_DATA.EDIT (%PROG);    -- .CURRENT avoids
                                   -- recursive call on script
  end EDIT;
```

```
--Compile script contained in a data object named "COMPILE"
  procedure COMPILE (%PROG:  in STRING := PARAM_OBJ) is
  begin
          if %PROG = PARAM_OBJ then
            GET %PROG -< PARAM_OBJ
          end if;
          %LIBNAME := %PROG & ".LIB"
          MAKE_LIBRARY (%LIBNAME)
          .CURRENT_DATA.COMPILE (SOURCE=>%PROG,
                                 LIB=>%LIBNAME);
  end COMPILE;
```

MAPSE COMMAND PROCESSOR

This section discusses the structure of the command processor.

Architecture Issues

The CP is a tool in the sense defined earlier; in particular, it is written in host-independent Ada, it accepts input from the (Ada) default input file, writes output to the (Ada) default output file, and is subject to the standard tool access, policy, and management controls discussed later. Its only "unusual" characteristic is that it is, typically, the program invoked on behalf of a user at Login. The CP consists of a collection of Ada subprograms, tasks and packages which cooperate in the performance of the command language interpretation function. These packages modularize the structure of the CP so that future maintenance of the MCL and CP will be straightforward, and so that the individual packages themselves can be utilized by other tools and general Ada programs. For example, the MAPSE Debugger is based on the CP.

Implementation

The Command Processor contains a front end (PARSE), which parses user-typed commands into a parse tree. Since MCL contains many Ada-like statements, this parse tree's definition is similar to Diana in nature. The Lexical analyzer is generated via the same maintanance tools utilized in generating the compiler's lexical analyzer. However, it must be modified to avoid look-ahead if the current token is a newline.

The heart of the CP is a parse tree interpreter (TREE_INTERPRET), which performs the actions specified by a parse tree. The ability within MCL to execute commands in the background or as co-routines suggests a task model in which several

command trees are simultaneously interpreted by distinct TREE_INTERPRET tasks. All global CP data (CP variables, e.g.) are shared by these TREE_INTERPRET tasks and must therefore be managed by tasks to maintain synchronization.

Chapter 7

TEXT EDITOR

The text editor will probably be the most frequently used MAPSE user-interface. As such, it should be responsive, convenient, and powerful though easily learned. A great deal of care should be exercised in selecting and designing editing features, not only to enhance productivity, but to ensure that users will like to use the MAPSE, i.e., not find it tedious, arbitrary, or frustrating.

It is for these reasons that we rejected unconventional approaches and based our design instead on an editor that has been shown to be successful, viz the EX/Edit editor, developed at the University of California/Berkley, and generally used on UNIX systems. We have a great deal of experience with this and other editors and feel that the human engineering evidenced in EX/Edit would be difficult, if not impossible, to surpass for the purpose of the MAPSE design.

Starting with EX/Edit as a model, the text editor is integrated as a MAPSE tool by being able to: accept standard or redirected input and produce standard or redirected output, respond to interactive commands as well as stored scripts, invoke any other MAPSE tool or KAPSE function from within an edit session and provide an Ada editing mode in which Ada "words" are recognized as lexical elements.

A full description of editor capabilities and interfaces is presented in the Text Editor B-5 Specification.

Chapter 8

COMPILER

The Ada compiler is the keystone tool in the MAPSE and, ultimately, the APSE. All Ada programs, including those which form the MAPSE, require translation by the compiler. Thus, the development cost of Ada programs depends on compiler performance. The performance of Ada programs depends on the quality of code produced by the compiler. The reliability of Ada programs depends on error diagnosis by the compiler and its support of debugging. The rehostability of Ada programs depends on the ability to retarget the compiler. These four characteristics: compiler performance, code quality, error diagnosis and debugging support, and retargetability, have driven the design of the compiler.

The compiler does not operate in isolation; it interacts with the user, the data base, and other MAPSE tools. The first section of this chapter describes the way the environs of the compiler have influenced the design. The second section of this chapter describes the structure of the compiler and the requirements which lead to that structure. The third section of this chapter describes the features of the immediate and future targets which have influenced the compiler design. The fourth section of this chapter highlights those Ada features which have received special treatment.

COMPILER ENVIRONS

The compiler does not operate in isolation; it interacts with the user, the database, and other MAPSE tools. This section discusses these aspects of the compiler.

The compiler user accesses the compiler through the program integration facilities, discussed in detail in a separate document. The program integration facilities control compilation and linking, library creation and initialization, recompilation, and the creation of missing library units. The compiler inputs, passed by the program integration facilities, include the following:

- The source text, which may come from a file or standard input.

- The Ada program library against which to compile and into which to put the results of compilation.

- Compiler options.

The compiler options (parameters) that can be specified include the following:

- VERSION, a version name for the compilation.

- USE_VERSIONS, which specifies version selection for referenced units (specifications, enclosing units, units specified in WITH statements, USE, INLINE and GENERIC bodies).

- DEBUG, which specifies where symbolic breakpoints may be set.

- OPTIMIZE, which specifies the level of optimization.

- LIST, which controls what listings, if any, are to be produced.

The VERSION and USE_VERSIONS options support a library that can contain a number of versions of the same Ada compilation unit. For example, the library might contain versions compiled for different target machines, separate debug and optimize versions, or versions for different target configurations. This feature is essential to avoid the costly duplication of whole libraries when related Ada programs share most of their code but differ in some respects.

The DEBUG parameter controls the placement of breakpoint information in the compiled code. DEBUG => ON causes the compiler to place debug information in its output so that the debugger may set a breakpoint before or after any statement. With DEBUG => OFF, the compiler does not place debug information in its output. As a result, breakpoints cannot be set at arbitrary statements, but only at higher level points such as subprogram invocations.

The OPTIMIZE parameter specifies the level of optimization the compiler may use. The levels correspond not to arbitrary sets of optimizations, but to the extent to which optimization is permitted to affect debugging. The following four settings are provided for this option:

CAN_MODIFY — The compiler must not move fetches or stores past statement boundaries; optimization is effectively restricted to in-statement optimization. The user can modify the value of a variable at a breakpoint and expect that value to be used in subsequent execution.

CAN_INSPECT — The compiler must not move stores past statement boundaries; optimizations which move expressions are permitted. The user can inspect a variable at a breakpoint and expect the value to be the most recent value of that variable.

ON — The compiler is free to move both fetches and stores past statement boundaries and to keep variable values in registers. The user may still inspect and modify variables; however, the code listing must be consulted to determine where this is meaningful to do.

OFF — No optimization is performed.

The LIST option controls which listings are to be generated by the LISTER phase. The OFF and ON settings respectively disable and enable all listings. The following settings may be used in any combination to enable specific listing features:

SOURCE — The Ada Source input is listed.

INCLUDE — The contents of files specified in the *INCLUDE* pragma is listed.

XREF — A cross-reference of symbols is listed. This includes the nesting level and statement number of the declaration and the statement numbers of all references to the symbol, and a call/use summary for subprograms and entries.

ATTRS — The attributes of each symbol are listed. If XREF is also specified these are provided in a single listing.

ASM — The assembly and machine code are listed for each statement.

STATS — The code and data size for each subprogram, the compilation time, and the number of statements are listed.

ENV — A cross-reference of external symbols is listed.

In addition to these outputs, errors, warnings, and notes are listed. The LEXSYN phase incorporates a two-level error recovery technique. It first evaluates which local repair (deletion, insertion, replacement) results in the fewest errors on scanning ahead a fixed number of symbols. If one of the repairs is effective, it is carried out. Otherwise, the second

level of error recovery is attempted. The stack is popped until it has a shift transition for a special error terminal symbol. Then action routines for these rules advance the input until a legal shift symbol or a special beacon symbol (e.g., semicolon) is found. In any case an error message is output.

The SEM phase discovers three major classes of semantic errors: ambiguous overloaded references, undefined objects, and incompatible operand types.

The back end phases only output warnings. FLOW detects constant computations which can cause exceptions. TNBIND detects local variables which are live at entry; that is, potentially uninitialized local variables.

The user can further control compilation through the pragmas discussed near the end of the "Compiling Ada" section. Taken together, the compiler options and pragmas give the user considerable control over the quantity and kind of processing, the quantity and kind of listings, and the interaction with the data base and other MAPSE tools.

The compiler, together with the program integration facilities, support Ada libraries as a composite object in the KAPSE data base. This is discussed at length in the program integration facilities. From the viewpoint of the compiler, the library exists at a single level, accessible uniformly through the Virtual Memory Management (VMM) system. The compiler need not concern itself with reading and writing information in the library. The VMM causes pointers to separate compilations to appear to the compiler as local pointer references. This vastly simplifies the compiler design. In addition, the VMM manages the in-core windows on the library so that the compiler may operate efficiently in smaller configurations, without arbitrary restrictions on program unit size.

The program integration facility addresses the optimization of recompilation, by postponing recompilations and avoiding redundant recompilation. This feature, described in detail in a separate document, is crucial to the overall performance of the Ada compilation facility. The compiler system will not only meet performance requirements expressed in lines per minute, but can entirely bypass whole compilations which the program integration facility determines are unnecessary.

The compiler interfaces with the linker and the debugger through the library. The compiler is not just designed for the MAPSE, however. It is designed to grow to support interfaces with other Ada tools. These interfaces are supplied by the Diana program representation in the library. Some of the envisioned tools which contributed to this design are the following:

- An Ada structured editor which directly operates on the Diana Abstract Syntax Tree (AST), bypassing LEXSYN.

- Any of a number of program analysis tools operating either on the AST or full Diana.

- Program transformation tools which augment Ada, operating either on the AST or full Diana. For example, a transformation which inserts code to collect the execution profile of a program.

- Tools to directly interpret Diana, after appropriate transformations. This would permit very high level debugging to be performed.

- Tools to perform functional simulation of ECS software on host computers. The debug "hooks" placed by the DEBUG compiler option permit run-time statement processor routines to gain control of execution, not only for debugging, but for a process control interface to environmental models, recording of variables, implementation of diagnostics, and modeling target code execution time. All this operates at full host speed while providing a high fidelity model of elapsed ECS computer time and real time interactions.

A compiler design that failed to consider the likely patterns of growth of the APSE would be a dead end design; this compiler design goes well beyond the MAPSE. We expect it to be able to support both tools we have envisaged and, by its flexible interfaces, stimulate the production of tools which have not yet been invented.

COMPILER STRUCTURE

The structure of the compiler has been chosen because it has been shown to work in a number of other implementations. This section highlights some of the features of that structure which are responsible for its success. In particular, this section discusses the choice of Diana as an intermediate language, the Virtual Memory Management (VMM) facility, the Front End/Middle/Back End structure, and the table driven nature of many of the compiler phases.

Diana

The intermediate language Diana was chosen for this design for a number of reasons. Technically, Diana is superior to the alternative intermediate languages that have been proposed for Ada; it combines the best features of its predecessors, AIDA and TCOL/Ada. Since Intermetrics participated in the design of both

TCOL/Ada and Diana and in the review of AIDA, we are familiar with the details of each. Among the major advantages of Diana are the following:

- Based upon the abstract syntax tree developed by the Ada formal semantic definition team;
- Applicable to a compiler back end, compiler front end (for separate compilation checks), and to other APSE tools (recompilation checker, text formatter);
- Definable in Ada;
- Efficiently implementable;
- Allows a human readable representation;
- Retains the structure of the original source text.

In addition, given the high probablility that Diana will become a commonly used interface for Ada tools produced by different contractors, it is the only practical choice for the MAPSE and compiler.

In the course of compilation, the Diana representation is transformed in a number of different ways. After LEXSYN Diana has no semantic attributes and is known as the Abstract Syntax Tree or AST, for short. After SEM Diana has a full set of semantic attributes. After EXPAND, the level of the language has been lowered to explicitly include the run time storage model. After VCODE the level of the language has been lowered to include features of the target instruction set.

Changes to Diana are of three forms: (1) Attributes may be added to nodes, as STORAGE does to represent storage allocation. (2) Tree transformations are applied which replace one set of nodes by another equivalent set of nodes, as in FLOW. (3) The underlying representation of Diana may be augmented to expedite certain kinds of processing, as when two-way links are added in the output of CODEGEN to expedite the processing by FINAL.

Virtual Memory Management

The Virtual Memory Management facility (VMM) is based on the LG facilities [Fostel 80] which have performed a similar function in existing compilers. VMM provides for the following capabilities (further details may be found in Chapter 10 of this document):

- Translation between internal and human readable form providing both for test input to phases and inspection of phase output.

- One level storage management of a potentially huge database.

- Clean compiler interface transparent to change of representation for efficient rehosting.

Structure

The organization of the compiler into Front End, Middle, and Back End logically separates different classes of processing:

- The Front End is target independent.

- The Middle selects the run time representation of Ada objects and expands references to objects to reflect these choices.

- The Back End optimizes and generates code.

Note that although some target dependent transformations occur in the Middle, so that they can be subject to optimization, this does not mean that 2/3 of the compiler is target dependent. Most of the Middle (GENINST, STATINFO, and large parts of EXPAND) are independent of the target. Furthermore, the target dependencies of the back end are segregated where possible in tables rather than in the algorithms which are driven by these tables. For example, TNBIND is target independent in all respects except for the tables which describe the classes and number of registers and the payoff functions which describe profitability. In short, expanding machine dependencies early does not increase retargeting costs, rather, they are both reduced and isolated.

It is important in many cases (e.g., during the initial stages of debugging a program) to minimize compilation time, with a possible sacrifice in run-time efficiency. This can be accomplished by specifying the optimization option to the computer as OFF. The effect is to obtain a "quick path" through the compiler, affecting EXPAND and the Back End phases. For EXPAND, the INLINE and MONITOR pragmas are disregarded, and no attempt is made to optimize away constraint checking (the SUPPRESS pragma is still obeyed, however). For FLOW, the internal form of a compilation unit is changed to an indirect Diana representation in Setup. No other FLOW transformations are performed. VCODE processing is unchanged. TNBIND processes a statement at a time; registers are not allocated globally. Each forced-CSE node is assigned a different temporary location. CODEGEN is unchanged. FINAL performs no peephole optimizations. For the 8/32 all branches are long branches; literals are pooled whether or not referenced. For the 370, code is segmented, but no attempt is made to choose the best boundary at which to segment. Literals are placed in the literal pool rather than in

the segment where they are referenced.

TARGETING AND RETARGETING

The compiler is designed to produce high quality code for the IBM 370 and Perkin-Elmer 8/32. This section discusses the design for the production of high quality code, including both the compiler and the run-time model. Beyond this, the compiler will be retargeted for other host machines and ECS target machines. This section also discusses the applicability of this design to machines other than the two initial hosts.

Code Quality

The compiler is conservatively designed; it uses state of the art techniques as opposed to unimplemented research. Notwithstanding, the list of optimizations it performs is impressive:

- Generic instantiation optimization under pragma control.
- Constraint checking optimization.
- Storage alignment.
- Inline expansion of subprograms, under pragma control.
- Task optimization, under pragma control.
- Literal pooling.
- Concatenation optimization.
- Local blocks folded into enclosing stack frame.
- Zero-trip-loop test optimization.
- Limited interprocedural analysis.
- Constant folding for standard functions, operators, indexing, selection, aggregates, type conversion, constraint checking, and change of representation.
- Constant propagation.
- Elimination of unreachable code.
- Elimination of common subexpressions.

- Movement of invariant code from loops.

- Induction variable elimination.

- Reduction of operator strength.

- Test replacement.

- Conversion of Boolean operations to transfer logic in control contexts.

- Algebraic simplifications.

- Transformations of if, case, loop, and exit when with constant arguments.

- Base register optimization (for 370).

- Special case code selection.

- Address mode determination which folds constants and additions into the effective address computation.

- Execution ordering.

- Global register and temporary allocation.

- Storage sharing of temporary results.

- Elimination of dead code, that is, computations whose results are not needed.

- Peephole optimizations for adjacent jumps and cross-jumping.

- Relative branch optimization (for 8/32).

- Code segmentation (for 370).

It may also be noted that there are a number of optimizations, commonly listed in the literature, that are not performed because the cost in compiler complexity cannot be justified by expected payoff:

- Loop unrolling is only done for loops known to be executed one or zero times. A more general form of this optimization may be easily added for targets for which it has higher payoff.

- Loop fusion requires analysis of data flow between array components referenced on different iterations. This analysis cost cannot be justified by the expected (small) payoff.

- Movement of invariant code from loops.

- Induction variable elimination.

- Reduction of operator strength.

- Test replacement.

- Conversion of Boolean operations to transfer logic in control contexts.

- Algebraic simplifications.

- Transformations of if, case, loop, and exit when with constant arguments.

- Base register optimization (for 370).

- Special case code selection.

- Address mode determination which folds constants and additions into the effective address computation.

- Execution ordering.

- Global register and temporary allocation.

- Storage sharing of temporary results.

- Elimination of dead code, that is, computations whose results are not needed.

- Peephole optimizations for adjacent jumps and cross-jumping.

- Relative branch optimization (for 8/32).

- Code segmentation (for 370).

It may also be noted that there are a number of optimizations, commonly listed in the literature, that are not performed because the cost in compiler complexity cannot be justified by expected payoff:

- Loop unrolling is only done for loops known to be executed one or zero times. A more general form of this optimization may be easily added for targets for which it has higher payoff.

- Loop fusion requires analysis of data flow between array components referenced on different iterations. This analysis cost cannot be justified by the expected (small) payoff.

- Hoisting and sinking of code are primarily space optimizations which require additional, costly analysis. In addition, the cross-jumping peephole optimization catches many of the common cases of this optimization.

The compiler back end phases share a number of design features that simplify the design, reduce the amount of compiler code, yield good code quality, and, as we shall argue later, reduce retargeting costs:

- Dataflow analysis is based on structured control flow.

- Payoff driven optimizations choose the most profitable alternatives both for code and register allocation.

The code generation for the 370 and 8/32 is based upon successful compiler designs for both of these machines by Intermetrics and COMPASS.

Run-time System

As important as compiler optimization may be, the choice of run-time system is as, or more, important to the run-time performance of ECS code. For this reason, the compiler design can be tuned to ECS requirements by pragmas. These pragmas control the management of storage, providing a range of options appropriate for different requirements. In the common case, a stack frame may be allocated static storage, reducing run-time storage management costs to zero. Other options include control of storage for accessed data which cannot be statically allocated but which must nevertheless be handled at very low overhead.

Retargeting

While the compiler will first be targeted for the IBM 370 and then retargeted for the Perkin-Elmer 8/32, we do not feel that this by itself is sufficient proof of retargetability. Because the two machines are so similar in word size, data formats, registers, and instruction sets, much of the back end that will be the same for these two machines will be different for, say, an ECS computer with shorter word length, different data formats, different register complement, addressing modes, instruction sets, etc. We know the basic design is appropriate for other target architecture because it has been used successfully in existing compilers for other machines (e.g., the 8086 and PDP-11). Moreover, the design is highly parameterized and table driven. Retargeting is accomplished, to the extent practical, by changing parameters and substituting different driving tables. The basic algorithms are unchanged.

The target parameters include the following:

- word size(s) and data formats for the storage allocation algorithm.

- register classes and numbers for TNBIND, the register allocator.

The tables to be changed include the pattern/transformation tables of EXPAND, VCODE, CODEGEN, and FINAL. EXPAND will be changed only in the part that deals with the run-time model; much of EXPAND is target independent.

The target payoff functions used by FLOW and TNBIND must be changed to reflect instruction size and costs for the new target. The FLOW and TNBIND algorithms are largely target independent, except for these payoff functions.

The tables in FINAL that drive the building of object code and the formatting of the assembly listing must also be changed.

COMPILING ADA

The compiler attempts not only to provide a correct implementation according to the language definition, but also to provide an efficient implementation of Ada language features. The compiler is designed to be efficient along the following dimensions:

- Size and speed of compiled code.

- Size and speed of the compiler and run-time system.

- Simplicity of the compiler.

The compiler is designed so that the implementation of an Ada feature does not cause run-time overhead for those who do not use the feature. For example, there is no time or space overhead for exceptions in a procedure that neither raises nor handles an exception.

For many Ada features, processing in the general case can be quite complex. The compiler, of course, must be prepared to handle such cases. However, the simpler case can be expected to be the norm and must be handled more efficiently. For example, optimization will handle programs with goto statements; however, processing is expedited by the reasonable assumption that goto's will be rare.

These principles have been applied repeatedly in this compiler design. The following paragraphs highlight the handling of specific Ada features. The order of topics corresponds to the

LRM presentation order. Parenthesized references give the pertinent sections of the compiler B-5 specification.

Lexical Elements

Lexical elements are handled by the LEXSYN phase of the compiler (3.2.2). The representation of literals is chosen by EXPAND (3.2.7.2a). Reserved words are detected by an exact hash function on each identifier (3.2.2.2c). This is rapid and simplifies the finite state machine tables that are part of the lexical analyzer. Both built-in and implementation-defined pragmas are supported. These are discussed near the end of this section.

Declarations and Types

The STORAGE phase (3.2.6) determines the representation of each type and object according to the run-time model (10.1). However, it may also be necessary to calculate the representations earlier, for cases where a storage attribute appears in a static expression (3.2.6.2). This is accomplished through a procedure CALC_REP, which can be invoked by SEM. The elaboration of declarations is accomplished in the EXPAND phase of the compiler (3.2.7.2b). A listing of declared symbols and their attributes is a product of the LISTER (3.2.13.2b).

Constants whose length is determined by a dynamic size initialization expression require the special handling discussed in (10.3.2).

The implementation of a derived type causes operations of the parent type to be inherited; the subprogram bodies are not copied (3.2.3.2h).

The overloading of enumeration literals is handled along with other overload resolution (3.2.3.2f).

For arrays, the component subtype descriptor is not replicated for each component (10.1.5). Static descriptors do not appear at run-time.

The dynamic parts of records are reached by offsets from the fixed part rather than by pointers (10.1.6). This is required to assure efficient "block assignment". A discriminant descriptor efficiently represents the tree of variants and speeds constraint checking (10.1.6).

The Ada limitations on access types limit the potential for aliasing which complicates optimization of other languages. Ada's limitations are used to good effect to simplify FLOW optimization (3.2.8).

Names and Expressions

The EXPAND phase makes the fetching of values from names explicit (3.2.7.2c). This allows the fetch operation to be separated from the addressing of the location. As a result, the addressing computation in a fetch context can be recognized as a common subexpression of a similar computation in a store context, e.g., A(I,J) := B(I,J)+1.

SEM (3.2.3) resolves overloading for indexed components and selected components. STORAGE adds a component_size attribute to each array node and a rep attribute to each record node (3.2.6.2b) in preparation for EXPAND (3.2.7), which generates in-line addressing operations for array and record references.

Packed array slices may require that both a byte address and a bit offset to be computed. This is an issue both for parameter passing (10.1.5) and for assignment (3.2.9.2a). In each case, the bit offset is only present when necessary.

Ada attributes, such as 'FIRST, are normally processed by the EXPAND phase (3.2.7(1)), which turns them into the appropriate data references. However, these attributes are interpreted in the SEM phase (3.2.3) when they appear in a context requiring a static value.

Overload analysis of aggregates is performed by SEM (3.2.3.2e). Aggregates are incorporated into overloading analysis in a way which makes them appear like calls to an anonymous procedure. The aggregates themselves are expanded by EXPAND (3.2.7.2c), which makes special cases of static aggregates, static constraints, array aggregates with an others choice, packed bit-string aggregates, and aggregates that initialize constants or variables.

The logical operators are implemented as control logic in control flow contexts (3.2.8.2g). Operators with constant operands are computed at compile time (3.2.8.2c). In general, operators are subject to special case analysis to generate efficient code (3.2.9).

Two kinds of compile time arithmetic are provided (3.2.3.2g) to handle both target machine arithmetic for constant folding and arbitrarily accurate arithmetic for static expressions.

Statements

In general, EXPAND (3.2.7.2d) expands code for statements. Later, VCODE (3.2.9) chooses efficient code for special cases. For example, the case statement will be coded differently depending on whether the alternatives are compact or sparse. As with other back end choices, the OPTIMIZE SPACE/TIME criteria are

applied. See (3.2.9) for a discussion of space/time tradeoffs.

The implementation of blocks yields zero execution costs at block boundaries. This is accomplished by incorporating local variables and temporaries in the enclosing stack frame (3.2.7.2d). This is possible even when the block contains an exception handler, due to the use of handler maps (10.4.2).

Assignment statements are expanded (3.2.7.2d) to make constraint checking explicit. Where, as in packed bit slice assignments, the starting position might not be on an even byte boundary, a bit offset is computed (3.2.7.2d). The assignment statement is also the subject of extensive case analysis in VCODE (3.2.9.2), where the length and alignment of the operands create special cases as does the range of lengths (if dynamic) and the relative alignment of the operands. Assignments that involve concatenation are optimized to eliminate the use of a temporary, if possible. In FLOW (3.2.8.2c), references to variables that are assigned constant values are replaced with the constant value. If this eliminates all references to that assignment, it will be eliminated by TNBIND (3.2.10).

If, case, loop, and exit when statements are processed at compile time for constant valued parameters (3.2.8.2). This effectively implements conditional compilation, since unreachable code that results is eliminated. For example, zero and one iteration loops are unrolled. The first case yields no code, the second dispenses with the loop logic. EXPAND (3.2.7.2d) places a zero trip test outside of each loop and places the loop termination test at the bottom of the loop. In the usual case, the zero trip test can be performed at compile time. In any event, the loop proper now will be executed at least once, assuring that FLOW optimization (3.2.8) will have a safe and profitable target location for invariant code it moves from the loop (3.2.8.2e) and for induction variable elimination and reduction of operator strength (3.2.8.2f).

The goto statement is restricted in Ada, so that no legal program contains a multiple entrance statement. This fact is employed in FLOW optimization (3.2.8) which processes data flow problems as a hierarchy of control structures (3.2.8.2b). This processing is efficient both in compile time and space, while achieving high levels of optimization.

Subprograms

Overload resolution for subprogram calls is done by SEM (3.2.8.2c). The expansion of subprogram calls and returns is handled by EXPAND (3.2.7.2d and e) following the parameter and stack frame convention of the run-time system (10.2.1 and 10.3). The conventions pass small parameters by copy and large parameters by reference. Small parameters include scalars, and

arrays and records less than two words in length. The first few parameters are passed in registers of the appropriate register class. The actual number of registers of each register class used for parameter passing is a compiler implementation parameter. Subtype descriptors are only passed for unconstrained array formal parameters.

The conventions for establishing and freeing stack frames (10.3) provide for exception handling and dynamic sized function return values, but not at the expense of procedure calls that do not involve either of these features.

The Ada calling conventions restrict the aliasing that may result from parameter passing. This is exploited by FLOW optimization (3.2.8) when it computes the effects of fetching and storing formal parameters values.

Packages

SEM (3.2.2.2b) produces separate symbol tables for the visible part and the private part of packages. This simplifies USE processing. Predefined packages are included in the parent scope of every library unit. EXPAND (3.2.7.2f) expands references to package data. Such data is referenced from the stack frame for the enclosing unit, which allows packages to be treated uniformly, whether library units or subunits.

Visibility Rules

LEXSYN (3.2.2.2b) associates a symbol table with each scope. This symbol table includes, by reference, the USEd symbol tables borrowed from other scopes. Because the Virtual Memory Management pointers to other compilation units are no different from other pointers, the symbol table lookup is simplified.

Renaming is processed by EXPAND (3.2.7.2g) so as to perform any necessary constraint checking. A store node is generated to save the address of the renamed object or component.

FLOW does a limited amount of inter-procedural analysis for procedures that are part of a single compilation unit. For other procedure references, FLOW makes a conservative assumption about what a given procedure can conceivably modify, based on visibility rules. For example, a call on a library procedure cannot modify the caller's local variables unless they are passed as in out or out parameters.

Tasks

EXPAND transforms tasking constructs into calls to run-time system routines (3.2.7(5) and 3.2.7.2h) according to the KAPSE specification. Rendezvous optimization occurs if the user specifies the pragma MONITOR(T) for a task or task type, T. This reduces the number of times the scheduler is called. This approach was chosen to provide a correct implementation and greater efficiency for a common special case without the compile time analysis required to determine when it is appropriate. Such an implementation provides an adequate facility for KAPSE implementation and can be extended when more ambitious tasking optimizations, e.g., [Habermann 80] are better understood.

Tasking requires that the run-time system support multiple stacks. To do this efficiently, the run-time system provides a dynamically managed tree of fixed-size contiguous data storage composed of stack segments (10.2.1). Each activated task acquires a stack segment from heap storage. Subsequent growth of a task's stack may require the allocation of one or more additional stack segments. Stack segments are freed as they become unused. The size of a stack segment is generally large enough to hold the stack frames of several procedures. In this way, the cost of heap management is minimized, for calls on the heap manager will be relatively infrequent. The fixed part of the stack and any single dynamic size value is contained in a single stack segment to allow efficient addressing of its contents.

Program Structure and Compilation

These features of Ada are largely handled within the design of the program integration facility, described separately. The LISTER (3.2.13.2.e) provides an environment listing. This listing gives a cross reference for each external symbol referenced in the compilation unit. This information is collected by STATINFO (3.2.5).

Exceptions

Exceptions are described in detail in (10.4). This feature is implemented with the goal of minimizing the cost of processing unexceptional code, at the expense of interpreting exceptions when they do occur.

Ada places restrictions on the extent to which code that can raise exceptions may be transformed [LRM 11.8]. The compiler is designed not to move code (3.2.8, 3.2.9) that could cause exceptions to a point where an exception would be handled by another exception handler.

When compile time computation of constant expressions raises an exception, the corresponding code is replaced by code to raise the exception (3.2.8.2c).

Generic Program Units

GENINST (3.2.4) is responsible for processing generic declarations, instantiations, and references to the instantiations. The OPTIMIZE(SPACE) pragma may be used in the generic part to control whether generic procedures with formal procedure parameters result in separate instantiations or a single instantiation with a procedure parameter. This choice of instantiation method is independent of the use of the OPTIMIZE SPACE/TIME pragma for the body of a generic procedure.

Representation Specifications

STORAGE (3.2.6) processes representation specifications. It determines the actual storage layout for all types and checks representation specifications for validity. EXPAND (3.2.7) translates references to data so that addressing is explicit. For example, EXPAND generates extra code for for loops with an enumeration type loop parameter with a representation specification.

Input Output

The compiler, per se, does no special processing for input output, short of accessing the predefined input output packages and mapping input output operations to references to a more primitive package. This mapping is performed by GENINST (3.2.5). The primitive input output operations are provided by the KAPSE.

Predefined and Implementation Defined Pragmas

LEXSYN (3.2.2) processes the INCLUDE pragma. The LIST(INCLUDE) compiler parameter controls the listing of included text (10.13.2a). The LIST pragma is also handled by the LISTER.

LEXSYN places the MEMORY_SIZE, STORAGE_UNIT, and SYSTEM pragmas in the Diana tree. These establish values for the configuration and machine dependent constants.

STORAGE (3.2.6) and EXPAND (3.2.7) process CONTROLLED and PACK. Additional control of storage is provided by the pragmas MARK_RELEASE and STATIC (3.2.7.2a and 10.2.2.3). Accessed data (10.2.2) fall into one of four categories depending on whether they reside in the stack or on the heap and whether storage reclamation is automatic or explicit. These categories of

accessed data are called Collection data, Controlled data, Checkpointed data, and Normal data. These choices allow the user to decide between time efficiency and flexibility of allocation/deallocation. These are described in detail in (10.2.2) and summarized in (10.2.2.5). Note that the design does not provide a built in automatic storage reclamation facility, such as a garbage collector. The provision of a garbage collector would add a substantial additional cost to the implementation, since the garbage collector would have to know the locations of all pointers. The overhead in time and space of garbage collection is generally incompatible with the requirements of real time programming in an ECS environment. The storage management options provided by this design have minimal overhead and while admittedly inadequate for artificial intelligence applications, are adequate to program both the MAPSE and ECS applications. Should a subsequent implementation support automatic reclamation as the default, all existing programs using the default allocation of this design would continue to run correctly.

The INLINE, INTERFACE, and SUPPRESS pragmas are processed by EXPAND (3.2.7). The optimization of constraint checking performed by EXPAND should reduce the need for the use of the SUPPRESS pragma in many cases.

The OPTIMIZE pragma is used by several compiler phases to choose between alternative program transformations. GENINST (3.2.4.2d) uses OPTIMIZE to determine the proper instantiation of generic subprograms with subprogram formals. EXPAND and back end phases use OPTIMIZE in the payoff functions used to compute the relative value of alternative transformations.

The PRIORITY pragma is taken into account by EXPAND (3.2.7) in its expansion of tasks.

Chapter 9

DEBUGGER AND EXECUTION CONTROL

INTRODUCTION

The MAPSE debugger design addresses the requirement for executing and debugging three classes of Ada software: (a) MAPSE and APSE tools which are intended to execute on the host, (b) embedded computer software (ECS), developed and verified on the host but intended to run on a target machine, (c) more general Ada programs (e.g., analysis, management tools, etc.) intended to run on the host. The selected design approach supports all three and lays a foundation for future comprehensive simulation and debugging facilities within an APSE.

It should be emphasized that the view taken here is that debugging and execution control (simulation) are in fact aspects of one activity, that of observing (or altering) the workings of Ada software while under execution.

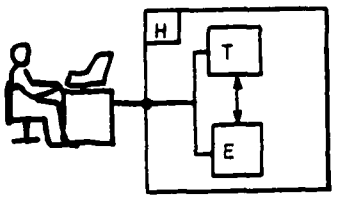
REQUIREMENTS

Figure 9-1 shows a variety of potential APSE operational configurations.

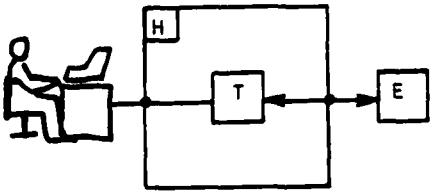
STONEMAN provides the following definitions and requirements:

> "An APSE adopts a host/target approach to software construction, that is, a program which will execute in an embedded target computer is developed on a host computer which offers extensive support facilities. Except where explicitly stated otherwise, this document refers to an APSE system running on a host machine and supporting development of a program for an embedded target machine."
> [STONEMAN, 2.B.2]

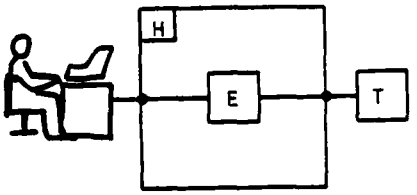
HOST/TARGET MACHINE/TARGET ENVIRONMENT CONFIGURATIONS*



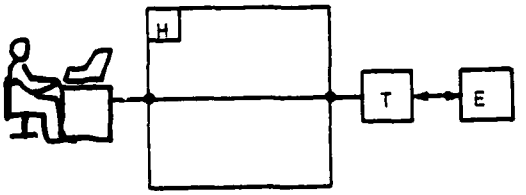
1a. C0: Host Emulated Target; Host Simulated Target Environment



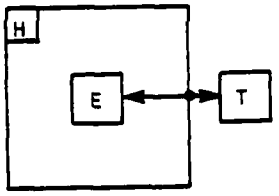
1b. C1: Host Emulated Target; External Target Environment



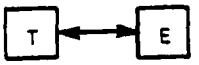
1c. C2: Host Connected Target; Host Simulated Target Environment



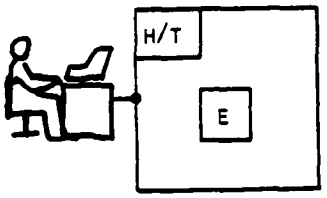
1d. C3: Host Connected Target; External Target Environment



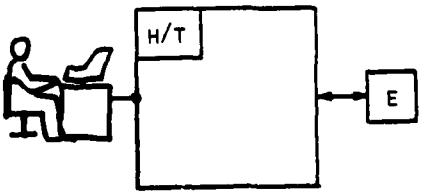
1e. C4: Disconnected Host; Host Simulated Target Environment



1f. C5: Disconnected Host; External Target Environment



1g. C6: Host and Target Same; Host Simulated Target Environment



1h. C7: Host and Target Same; External Target Environment

*Fairley: "Ada Debugging and Testing Supportive Environments"

Fig. 9-1

"An APSE shall permit testing and debugging of an Ada program executing in any target machine supported by an APSE. It shall be permitted for such a program to use the machine-dependent features of the language. The facilities for testing and debugging of target-resident programs should be based upon the equivalent facilities for host-resident programs."

[STOLMAN, 4.E.10]

Based on the above and the minimal APSE objectives of the S.O.W., the current debugger/controller design encompasses (or provides a basis for) configuration C6: host and target same, i.e., for the development and checkout of MAPSE/APSE tools and other general Ada programming, and configuration CO: host emulated target, i.e., for the development and checkout of ECS software intended to run on a target machine. Two other configurations influence the design, viz C2 and C3. In these cases, ECS software runs on an actual target machine being controlled and/or supported by a host and either simulated or real environments. The design philosophy accounts for C2 and C3 but specific features are considered outside the current scope of work.

The MAPSE debugging tool, in operation, establishes through KAPSE facilities a "controller/controlled" relationship with any executing Ada program. That is, the debugger may start, stop or modify a controlled program. Using this mechanism, the proposed design meets every requirement of the SOW. Specifically, the debugger will be able to activate breakpoints:

- before or after any Ada statement
- at a statement label
- after every statement (single stop) or after every n statements
- on exceptions
- on entry and return from subprogram units
- based on problem time (this is a suggested future APSE tool, see discussion below).

At a breakpoint, execution is suspended and control is transferred to the debugger to accomplish any of the following actions:

- set Ada variables
- modify flow of control by allowing execution to continue at a particular statement or label, or causing

return from a procedure or function

- raise an exception
- single or multiply step the program
- set conditions to modify or activate other breakpoints
- exercise KAPSE interfaces (this is a suggested future APSE tool: see discussion below).

In addition, at any breakpoint the debugger can inquire about the executing program, i.e., it can:

- display Ada variables
- trace procedure, statement or task history
- change scope to view any variables
- dump any and all variables in active scopes
- list the scopes and breakpoints.

The debugger syntax to accomplish the commands and actions listed above is implemented as extensions to the basic MAPSE Command Language (MCL) and therefore the debugger has at its disposal the MCL at any breakpoint. As a consequence of the integrated approach, the user can bring to bear the full power of the MAPSE while debugging, i.e., activate any MAPSE tool or facility in support of this objective. This can be effected either interactively or from stored scripts in a batch processing environment. In batch the breakpoints are set up in advance by commands from a script file. When a breakpoint is hit, the predefined breakpoint actions will be executed and the program will resume. Output will be directed to a file.

DESIGN CONSIDERATIONS

Successful source level debuggers have been written for other high level languages, and these can provide useful models for an Ada debugger. However, the Ada language and the MAPSE environment differ from others in two ways that have a direct effect on the debugger:

a. Ada supports multitasking. Thus, when a single program is "run", a number of Ada tasks may be active concurrently.

b. Ada is designed to support embedded computer systems (ECS), and many Ada programs will be targeted to special machines with limited resources.

Much of the functionality of the MAPSE debugger is typical of a well-crafted source level debugger. However, the fact that Ada supports multitasking requires special consideration. Classical debugger design has been based on the conceptual framework of a single task executing a program. However, Ada programs may involve multiple tasks executing the same program code. This requires that some concepts which were well-defined in a single-tasking environment must be carefully extended in order to remain valid in a multitasking environment.

In a single-tasking environment, a debugger can set a breakpoint at a particular point in the program. When program control reaches a breakpoint, program execution is suspended and control is passed to the debugger. The user is then able to examine and modify data. Finally, the suspended program is resumed, perhaps at a different point in the program. The context of this suspension is clear and consists of two parts. First, there is the static scope for the particular point in the program where the breakpoint was set. Second, there is a designation of the state of the program whose operation was suspended. This state consists of the subprogram call chain, that is, a listing of those points where the virtual machine saved its context and began to execute a subprogram and where control has not returned yet from that subprogram.

In a multitasking environment, the context of a suspension must be extended. First, when a task hits a breakpoint and control passes to the debugger, the user is told which task was suspended. Thus, the context of the suspension must be (task_id, breakpoint scope, call chain). When one task hits a breakpoint and control passes to Debug, other tasks are frozen. This guarantees that the Debug user will be examining and modifying a static environment. When control is returned to the program by user commands to Debug, all tasks are unfrozen and the program execution resumes in its possibly modified state.

The interface between the MAPSE debugger and the Ada RTS has been carefully set up to minimize impact on the Run Time System and the running program it controls. That is, the Run Time System provides a few basic elements of control and information transfer, and Debug does the rest of the work. The RTS actions are automatic and simple. It is Debug that maintains a data base and recorded commands. This clean separation of Debug Support Functions and Debug Processing allows easy extensibility of Debug capabilities with minimal effect on the Ada Run Time Environment.

"Hooks" were chosen to implement the concept of an Execution Control Point" because of their simplicity, efficiency, and

extensibility. Hooks have a minimal impact upon the Ada program both in execution time and program space, yet provide Debug with full control over the program being debugged. The hook information table compiled by the Compiler and Link Editor provide a place to store additional information in the case of APSE extensions. Additional information concerning the execution time or hardware side effects of each statement could be made available to Debug or other APSE tools by having them recorded in the hook table.

The "hook approach" directly addresses the tradeoffs between interactive debugging and batch verification testing. Considerations of the differing characteristics argue in favor of implementing breakpoints by having hooks placed in the object code by the compiler at every point where a breakpoint could occur. These hooks in a sense instrument the code and are implemented with great efficiency, causing but a small increase in running time, not noticeable by the interactive debugger. However, such instrumentation proves to be more efficient than the traditional interrupt approach for large scale verification testing, which requires more frequent context switching. Instrumentation provides the highest level of control at the lowest cost. The density of hooks in a block of object code determines how closely the execution of that code can be monitored.

The programmer can, through the use of the compiler parameter DEBUG, select one of two levels of hook density. With DEBUG OFF, the compiler will insert hooks only at subprogram entry and exit. This minimal level of instrumentation consumes negligible resources but provides enough information in most cases to give at least a rough indication of where and when an unexpected error occurred. With DEBUG ON, the compiler will place hooks between each statement in the compilation unit. This allows very fine control over program execution and observation. It is anticipated that most users of the MAPSE system will choose to compile most programs with DEBUG ON. A program version which has been tested and released for use on a system where resources are tight might compile with DEBUG OFF. In such a mode the debugger will still be able to display the subprogram call history and display or modify program variables at subprogram calls and returns. This should be more than adequate to make an initial analysis of any problem that should arise and to isolate the problem. If more debugging is required to fix the problem, a test program can be generated by recompiling selected parts with DEBUG ON. Experience with this compiler-supported instrumentation technique indicates that the penalties are small for leaving the hooks in, for programs intended to execute on the host. What is gained is control and debug flexibility.

The Ada compiler is capable of applying a wide variety of optimizations to the code being generated. Some of these optimizations are all but invisible, other than having the effect of increasing execution speed. However, many of the more powerful

forms of optimization unavoidably interact with the debugger, for they cause the program to execute operations in an order other than that indicated by the source program.

In order to provide the user with some control over this interaction, the compiler has an OPTIMIZE parameter which can take any of four values: CAN_MODIFY, CAN_INSPECT, ON and OFF. A value of CAN_MODIFY will allow all optimizations which permit the user complete use of all debugging commands. A value of CAN_INSPECT allows optimizations which will not affect the debugger's ability to display values but which may cause unexpected results if a user attempts to modify a value. For example, the program might evaluate a common subexpression once and store its value in a temporary variable. If there were a breakpoint between the evaluation of that expression and its use and if the user modified a variable on which that expression depended, the modification would not affect the value of the expression, as the user might expect. An OPTIMIZE value of ON allows optimizations which change the order of operations such that even displayed values may be misleading. A value of OFF will prevent all optimizations.

The debugger knows the optimization level for each compilation unit and will issue a warning if the user attempts to execute a debugger command which is inappropriate for that level.

DESIGN DIRECTIONS - FUTURE APSE TOOLS

Functional Simulation

The MAPSE debugger lays the foundation for a future MAPSE tool, that of Functional Simulation (FSIM) used so successfully for software development for the Space Shuttle [Intermetrics 80]. In essence, since machine-independent Ada is implemented for both host and target computers with identical semantics, the same source code has the same meaning on both machines. It therefore becomes possible to simulate and develop embedded computer software on the host computer. This is the intention of Stoneman definition 2.B.2.

The statement hook described above and shown in Figure 9-2 permits run time system routines to gain control of the execution. Aside from performing the debugging requests previously discussed, such control can advance a pseudo-clock to maintain problem time, act on and react to Ada tasking functions and interface to KAPSE or other support routines for environment updates, connections to systems outside APSE, etc. The pseudo-clock, really just a software quantity, is based on weighting the Ada statements with elapsed time that would have occurred on the target machine. Such times are automatically computed and

entered by the compiler in a cost table based on target machine characteristics. (Note that if host and target are the same, for example, MAPSE tools, then pseudo time becomes actual time and no computation is necessary - the actual clock can be read.) Although accuracy is variable, it is usually sufficient to support the desired level of verification. The opportunity now exists to develop an entire digital closed-loop simulator based on pseudo-time. That is, all dynamic occurrences, including Ada tasking, diagnostics and environment model updates, are performed consistent with this time. The objective of this simulator is to check out the FCS software to the greatest extent possible on the host facilities before commitment to the target machine and its more diagnostic-limited "hot bench" environment.

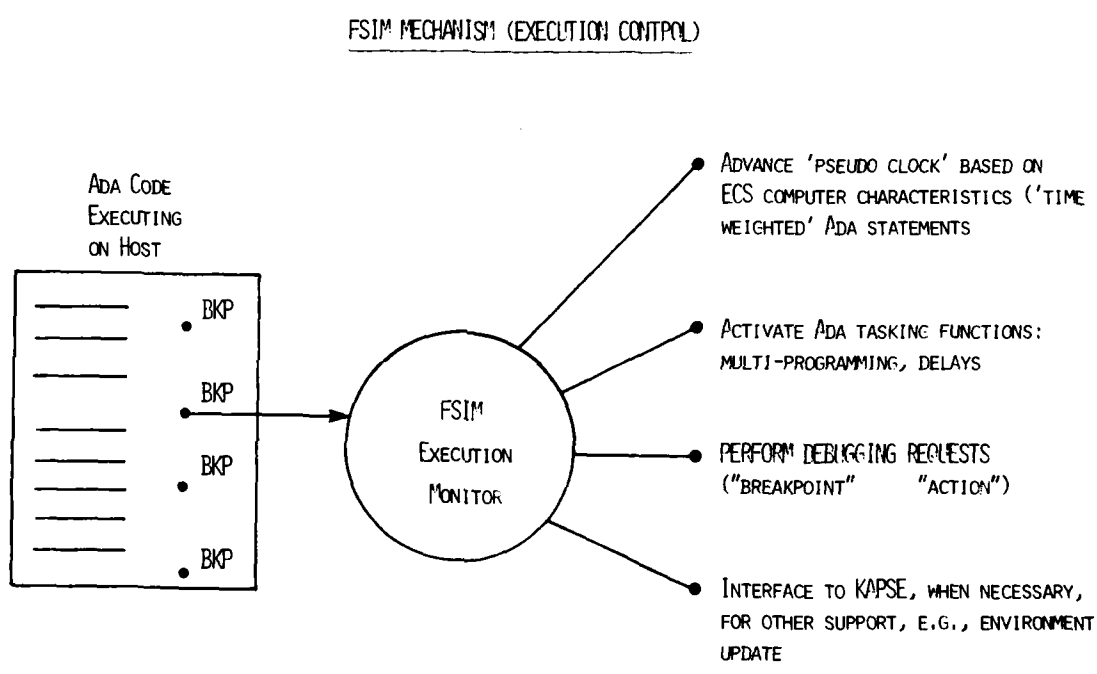


Figure 9-2

With the hook technique already in the MAPSE design, FSIM is a straightforward extension and will result in an effective and efficient statement-by-statement Ada "functional simulator" executing at full host speed (i.e., not an interpreter) while providing a high fidelity model of elapsed target computer time and real time interactions.

Debugging on the Target Machine

Debugging on the target computer itself is certainly desirable. To the extent that memory and resources are available, and an Ada run time system provided, the MAPSE debugger could conceivably be re-hosted onto the target machine. Practical considerations, dealing with symbol table sizes, available peripherals, etc., might require a two-machine set up where the target is connected to the MAPSE host (see Figure 9-1, config. C2) or to another support computer.

In either case, the intention of the design is to provide the same functionality of debug constructs and approach regardless of whether the ECS software executes in FSIM mode on the host or in situ on the target. The model of the debugger controlling an Ada program by communicating (through the KAPSE) with a debugger support routine extends naturally to ECS target debugging. A small, target-resident communication routine handles the inter-machine protocol with the host-resident KAPSE. The debugger design is essentially unchanged; the KAPSE program control functions must be extended to the target machine. Further designs or implications of target debugging are considered beyond the scope of this current effort.

Chapter 10

MAPSE GENERATION AND SUPPORT

Two classes of tools are employed in the generation and support of the MAPSE.

1. Generators. These tools construct pieces of MAPSE tools. A generator is used each time the corresponding tool is revised. The parser and lexer generators [MAPSE Generation Support (MGS) B-Spec, 3.2.1] create the LEXSYN phase of the compiler [Compiler B-Spec, 3.2.2] from formal grammars. Similarly, the VMM Representation Analyzer [MGS B-Spec, 3.2.2.4] takes an Ada description of the kinds of objects which can appear in a virtual memory, and produces the package interface to the VMM implementation. This interface package is included in MAPSE tools that utilize the VMM.

2. Bootstraps. These tools provide a means of getting the MAPSE "off the ground." Once the MAPSE is established, these tools are no longer needed for support of the MAPSE. The most important bootstrap tool is the Bootstrap Compiler [MGS B-Spec, 3.3].

The design must address different questions for these tool classes. For generators, should they be included as part of the MAPSE, i.e., should they be written in Ada? The harder question for bootstraps is how will they be realized? The following subsections discuss the reasons for the design choices that answer these questions.

GENERATORS BELONG TO THE MAPSE

Our design specifies that not only are MAPSE tools written in Ada, but their generators are written in Ada as well. This decision was based on several general principles.

In the first place, the SOW requires that the MAPSE be portable [S.O.W., 4.1.1.2]. The easiest way to achieve this goal is for the MAPSE tools and tools necessary for MAPSE generation to be written in Ada, and for these generators to be part of the MAPSE. If only the MAPSE tools were ported to a new host, it would be awkward, or at worst impossible, to make any revisions to the MAPSE on the new host. Instead, changes could only be

performed where the generators were supported, which in this case would be 370 CMS. This would be an unacceptable situation.

The second argument in favor of this approach appeals to the fundamental rationale for having an AIE; the AIE provides support for the development and maintenance of Ada programs. MAPSE tools are Ada programs, and therefore perfect candidates for being self- supporting. Some level of extending, adapting, correcting, and perfecting of the MAPSE tools is anticipated after the AIE is complete. This can only be accomplished if all of the tools (including the generator tools) are included in the MAPSE.

Independent of the MAPSE issue, Ada is an excellent language in which to write the generator tools. Ada supplies facilities that are as appropriate for building tools as they are for building embedded applications. It will also be a great convenience to decommission the 370 CMS bootstrap system early in phase two, and shift to a single development system of a higher quality (the MAPSE itself). As described below, the Bootstrap Compiler, implementing all of non-tasking Ada, will be available very quickly. This allows the generator tools to be built in time for use in developing the production compiler.

TOTAL SELF-HOSTING RATIONALE

MAPSE development work will be transferred from the CMS bootstrap system to the preliminary MAPSE environment well before the MAPSE is complete. This decision was easy to make even though there are strong arguments on both sides.

The chief argument against self-hosting the last of the development work is that to do so complicates the (already challenging) AIE development project. It increases the number of MAPSE tools that are needed, and adds several lines to the PERT chart.

Development costs are not seriously affected by the decision. The cost of additional tools is offset by the elimination of the need to maintain the CMS and AIE systems in parallel.

The compelling arguments for self-hosting are reliability, quality, and life-cycle cost considerations. Reliability depends on adequate testing. Our experience with compilers and other complex systems is that the first "real users" attempting to use a system for its intended purpose will always find bugs that escaped formal testing. Since MAPSE development is an intended "real" use of the MAPSE, we can be our own guinea pigs. Early self-hosting is a way to include user-testing within the original AIE schedule.

Self-hosting will also tend to increase the quality of the MAPSE/ KAPSE. As the first users of the MAPSE, we will be the first to discover any shortcomings, inconsistencies, and

"annoying little quirks" that may be in the design. We can identify the "microtools" that are needed to fill in minor gaps. These microtools on other systems are generally 2-10 line command language macros or "procs" that are invented over and over by users. For example, most 360 systems have dozens of functionally equivalent "microtools" for creating and submitting Job Control Language for printing a file on a line printer. By being the first real users, we can build any such missing capabilities either as shell scripts, Ada programs, or enhancements to the command language processor or other tools. The final, and clinching, argument for self-hosting the last development phases is that more effort will go into retargeting, rehosting, extending, and enhancing the MAPSE than will go into its original development. A small increment of effort in MAPSE development will ensure that it is a highly cost-effective tool later.

BOOTSTRAP ADA COMPILER

Two crucial decisions were made in arriving at a design for the Bootstrap Compiler:

1. The existing DARPA Ada Compiler, implemented by Intermetrics in Simula on the DEC System 10/20, will be moved to the 370 (where it will be supported by CMS Simula) instead of implementing some subset Ada compiler from scratch.

2. A new code generator will be added which produces PL/I as the target language, rather than assembly code.

These decisions are justified below on their own merits, and in relation to the above-mentioned design alternatives.

We also demonstrate below that the proposed Bootstrap Compiler has these properties:

- It will be implemented quickly and with a minimum of effort.

- It will compile and generate code for full Ada excluding only tasking features.

- The resulting compiled code will be reasonable in speed and size; i.e., the Bootstrap Compiler will be an effective tool for bootstrapping the Ada Compiler.

Use Existing DARPA Ada Compiler

Two usually conflicting criteria for a bootstrap compiler are that it take very little time to build, and that it compile a large enough subset of the language to make writing the Ada-in-Ada compiler relatively painless. The conflict is that providing a large subset usually takes a long time.

If one were to start from scratch, this would be particularly true of Ada. Facilities such as derived types, packages, generics, and separate compilation, which are very desirable to exploit in implementing the Ada-in-Ada compiler, are exactly those features that take the longest to implement in the bootstrap compiler. However, if the bootstrap provides only the barest minimum support, then the Ada-in-Ada compiler will be difficult to implement; and many of the bootstrap-required contortions will remain in the production compiler. Furthermore, few of the other MAPSE tools could be started, since they might need Ada facilities not supported by the bootstrap.

Instead, the strategy starts with an existing Ada compiler that supports the entire language. (The added code generator will not implement tasking.) Thus, from the start, the Ada-in-Ada compiler (and the other MAPSE tools) can be implemented in terms of Ada's best and most advanced structuring facilities.

Secondly, the Bootstrap Compiler can be brought up on the 370 very quickly. Not only does the compiler exist, but moving it from DEC System 10/20 Simula to CMS 370 Simula is very easy. In his paper on transporting Simula programs from the DEC-10 to the 370, Orgass writes:

> "With minor exceptions described below, the languages implemented by the two compilers is the same and this makes the transfer of SIMULA programs very easy. In contrast, it is very difficult to transfer Fortran programs from a DEC-10 to CMS." [Orgass 79]

The exceptions he refers to are text replacements such as substituting parentheses for square brackets.

Adding the new code generator to the DARPA compiler requires an intimate knowledge and understanding of the original compiler and of Simula. Intermetrics is uniquely qualified to perform this aspect of the bootstrap implementation.

Thus our approach will provide a bootstrap compiler that is available early, and that supports all of non-tasking Ada. This will have a very positive effect on the development of the production compiler and the rest of the MAPSE.

This strategy yields a further bonus: the Bootstrap Compiler is a 370 hosted Ada compiler that generates code for the 370, and it will be available well before the production compiler. Unlike the usual quick-and-dirty bootstrap compiler, this one will have the same user-friendly interface as the original DARPA compiler. This makes the Bootstrap Compiler appropriate for educational usage. It might also be desirable to have some low-risk projects start with the Bootstrap Compiler, and then

graduate to the production compiler when it is released.

Generate PL/I as Target Code

The combination of back-end and run-time system of any Ada compiler, including a bootstrap Ada compiler, must supply the following facilities:

- data type representation,
- stack layout and stack-frame allocation,
- heap management (accessed objects),
- subprogram call and return,
- 370 code selection,
- register allocation, and
- I/O.

These in fact are the minimum facilities needed in which to write the Ada-in-Ada compiler.

If it were decided that the bootstrap compiler generate assembly code, then a compiler back-end and a run-time system would have to be written that implement all seven facilities. Implementing all of these would require a great deal of resources, both in terms of people and in terms of schedule time. Furthermore, little of the back-end or run-time system could be salvaged for the production compiler (even with re-writing into Ada). For instance, the code selection strategy would be simple, and sub-optimal. The register allocation approach would use only a small, fixed set of registers. The stack layout decisions would omit considerations of tasking, etc.

The proposed design instead exploits a high level language, PL/I, as the target language for code generation. The Bootstrap Compiler's code generator performs a simple translation of the internal Ada tree to PL/I. Then the generated PL/I program is compiled by the 370 PL/I compiler. In this way, the PL/I compiler and run-time system provide all seven facilities.

It is not necessary to build a run-time system for the Bootstrap system. In fact, the Bootstrap Compiler only has to support one of the facilities - data type representation - and then only to a small extent because PL/I does not have type definitions.

The oft-expressed criticism of PL/I - that it has too many features - was the major reason for choosing it. In most cases, the translation from Ada to PL/I is straightforward; most Ada

constructs have an obvious mapping onto some PL/I construct. (Note that the advanced visibility and scoping features of Ada that are missing from PL/I such as packages and overloading, have effects only in the semantic analysis phase, and do not appear in the generated code.) For example PL/I has records, arrays, and access types (based variables); it even supports aggregate assignments of whole records and arrays. PL/I has ON conditions, which can be utilized to support Ada exceptions. Another important point is that PL/I provides separate compilation; this would otherwise be difficult to supply in a bootstrap system.

Other reasons for choosing PL/I are that it is vendor supported, and that the compiler contains an optimizer. This latter point can compensate for any sloppy PL/I code produced in the translation process.

SYSTEM GENERATION

It was decided that the most thorough, practical, and cost effective approach to system generation (tailoring the 370 system to other 370s or the 8/32 system to other 8/32s) was to rely on the facilities of the underlying operating system. The procedure is to do system generation for the OS/32 or the 370 VM operating system and then to install the unmodified MAPSE on top.

VIRTUAL MEMORY METHODOLOGY

Over the past several years compiler implementations at Intermetrics have been based upon a virtual memory strategy that allows compiler phases and other tools to operate on arbitrarily large program representations in a transparent manner, independent of host memory constraints. Such a facility is critical for Ada, because the separate compilation facilities require the preservation of large amounts of symbol table information. It cannot be assumed that the required external symbol data will fit in main memory during compilation. Virtual Memory Methodology (VMM) for the MAPSE draws upon our experience with previous implementations and is designed to meet several key MAPSE objectives: efficiency, reliability, portability and extendability. In the remainder of this section, we illustrate how these goals are met and describe the tradeoffs that were considered in making the design decisions.

Space Efficiency

The two principal considerations for space efficiency are the size of nodes and the size of virtual memory pointers ("locators"). Since the representation of a node is derived from an Ada type declaration provided by the tool builder, VMM allows a high degree of efficiency. The tool builder decides whether different nodes are variants of a single type, or whether they are different types, based on expected usage patterns and knowledge of the attributes. Minimal space overhead results from VMM

itself - from past experience, this is roughly two machine words per node. Locators will occupy either one or two machine words.

Time Efficiency

A critcal aspect of time efficiency is the amount of processing required to dereference a locator. One possible technique that minimizes this time is to perform an "environment read" step before processing begins. Basically, this step "links" VMM files together by replacing locators with more direct references (access values, if the files can be core resident). In this way, the dereferencing operation is inexpensive, but adds significant costs in other areas: (1) the linking is done even though only a small number of actual references may be needed; (2) a great deal of copying is required; (3) the technique loses its advantages when the files cannot be core resident. Instead of an environment read, VMM performs dereferencing of locators as required during the processing. Although this appears to introduce overhead for each dereference, in fact there are facilities under both system and user control that make this technique efficient. As an example, the user may "freeze" a locator (thereby obtaining an Ada access value as a node reference) and subsequently "unfreeze" it.

Another important aspect of time efficiency is the amount of recompilation that is required. As an example, inserting a comment in a compilation unit Q, and then recompiling Q, will result in a Diana representation that may have nodes at different positions than in the earlier version. A compilation unit depending upon Q may have locators into the earlier version that are no longer valid. It would be unfortunate if the dependent had to be recompiled because of out-of-date locators, since the addition of a comment should not invalidate dependents. To solve this problem, VMM permits a program to specify a locator translation with the following effect. When a locator is dereferenced and found to refer to a database object for which a translation is defined, the locator value will first be "looked up" in a table and a corresponding locator for a different database object will be substituted automatically.

Reliability

VMM enforces a centralized definition of the interfaces that are used between tools, thus ensuring at all times a single consistent view. This is important for reliability, since the job of system integration is made easier. As an example, the various Diana versions that serve as compiler phase interfaces are defined together; thus the system can ensure that an attribute required by one phase is in fact output by its predecessor.

The representation analyzer is the element of VMM that helps achieve this centralization. It also promotes reliability in that it automates the generation of data structure creation and

access routines and internal-to-external conversion routines based on the type declarations supplied. Note that the support of an external (ASCII) representatiion is itself important to reliability, since it aids the testing of the MAPSE tools.

Portability

There are two main characteristics of VMM that help achieve MAPSE portability. First is the virtual memory management, which allows tools to be written independent of host machine memory limitations. Obviously, working set limitations may degrade system performance on small hosts, but there is no need for the tool builder to be concerned about this level of detail in the writing of the tool.

A second portability consideration is the reliance on Ada as the metalanguage for defining the node structures. Although alternative formalisms were examined (e.g., *IDL* [Nestor 81]), using Ada offers several advantages:

1. It is sufficiently high level to be a readable notation.

2. It is sufficiently low level to permit efficient implementation.

3. It avoids a new notation and the requirement for a new translator.

Extendability

The MAPSE is expected to grow as new tools are introduced, and VMM easily supports such growth. Basically, a tool may define a new set of node types and create and reference objects from both the new types and existing types defined by another tool within the same domain. The addition of a new tool does not require recompilation of any existing tools.

Chapter 11

CONCLUSION

The design presented in the A and B-5 specifications and elaborated upon in this document meets the spirit of Stoneman and the requirements of the S.O.W. The high degree of integration among its elements both in design and conceptional use breaks new ground. The MAPSE virtual operating system (KAPSE) provides central management, communications and control. It provides facilities to organize projects, delegate authorities, distribute responsibilities, provide communications links among tools and users, and maintain data integrity in terms of configuration management.

The approach to specific MAPSE tools is one which keeps considerations of the user-interface uppermost. The APSE will not succeed in improving productivity, or even win acceptability, if it is perceived as difficult to use, cumbersome or inefficient. MAPSE tools have been carefully designed to support the stated objectives of Ada, that of developing embedded computer software. The design makes provisions for and looks to future APSE tools which will be required for ECS (target) development.

No feature has been proposed which cannot be realized or cannot be implemented efficiently in Ada. We believe the best state-of-the-art techniques are employed in the compiler design and eventual construction. The level of achieved optimization is more than adequate while still maintaining the conservatism necessary to ensure reliable code. Through the pervasive use of Ada and the isolation of machine dependencies within the KAPSE (and compiler code generation), rehosting and retargeting of the MAPSE will be a straightforward task of limited effort. Our bootstrap approach is of low risk and will allow rapid development of the system on the IBM 370/VM.

In summary, our design meets the requirements, is modern in approach and does form the basis for an Ada Integrated Environment.

APPENDIX

A.1 A Simple Programming Scenario

A.2 A Short Management Scenario

Appendix A.1

A Simple Programming Scenario

```
Logon AFAda    -- Host may dictate logon syntax

:edit example -- Create a new text file

>insert
-- This example program writes the string "Hello"
-- on standard output.
With Text_IO; Use Text_IO;
Procedure Example is
   Begin
      Put("Hello");
   End Example;
.
>write

>quit

: -- The dot '.' above is not part of the text of example.
: -- It told the editor to leave insert mode.
: -- The write editor command saved the text in "example"

:compile example proglib -- compile into new program library

PROCEDURE EXAMPLE COMPILED WITH 0 ERRORS.

:link proglib          -- Main unit assumed

:proglib.link.example -- Invoke program from library

Hello

:copy proglib.link.example my_search_lib.example

:                      -- Install program in library which

:                      -- is searched for commands

:example               -- Invoke program from user's library

Hello

:logout
```

Appendix A.2

A Short Management Scenario

Problem Statement

A manager begins a new project -- to develop a cockpit system with a stick control monitor, a heads-up display, and a keyboard and display command unit. The manager creates a composite object in which the various test levels of the system source, documentation, and test results will be collected. She then creates and sends out appropriate windows on the composite object to Jane, the developer for the stick monitor module, and George, the tester for test level one of the entire cockpit system.

The manager's first action is to create the composite object named COCKPIT, with four distinguishing attributes.

```
: CREATE_COMPOSITE  COCKPIT,
    COMPONENT_DA=>"MODULE  SUB_MODULE  TYPE  TEST_LEVEL"
```

The manager then specifies the categories of the components of COCKPIT. The modules names are pre-specified, but the submodules are left open; each module has an associated test level; each named module has a source part, a documentation part, and test results. She specifies three capacities (TESTER,DEVELOPER,MANAGER) and specifies for the types SOURCE and TEST_RESULTS the level of access control for each capacity.

```
: SET_CATEGORY COCKPIT      -- Fill in the detailed category def.
(CATEGORY_CLASS=>COMPOSITE --  for the cockpit composite object.
 COMPONENT_DA=>             -- Specify the distinguishing attributes.
    (MODULE=>(CONTROL_MON,HEADS_UP_DISPLAY,KB_COMMAND),
     SUB_MODULE,            --  (no limitations on SUB_MODULE name)
     TYPE=>(SOURCE,DOC,TEST_RESULTS),
     TEST_LEVEL=>(0..*)
    ),
 COMPONENT_CAPACITIES=>(TESTER,DEVELOPER,MANAGER),
 COMPONENT_CATEGORIES=>
    ((TYPE=>SOURCE)=>       -- All components with TYPE=>SOURCE...
       (CATEGORY_CLASS=>SIMPLE,
        CATEGORY_NAME=>ADA_SOURCE,
        ACCESS_CONTROL=>
          (TESTER=>(READ,COPY),
           DEVELOPER=>(ALL),
           MANAGER=>(READ)
          )
       ),
     (TYPE=>TEST_RESULTS)=> -- All components TYPE=>TEST_RESULTS...
       (CATEGORY_CLASS=>SIMPLE,
        ACCESS_CONTROL=>
```

```
            (TESTER=>(ALL),    -- Tester controls TEST_RESULTS.
             DEVELOPER=>(READ),
             MANAGER=>(READ,COPY)
)   )  )  )
```

Now the manager creates a window on COCKPIT with the capacity DEVELOPER, but limited to the module CONTROL_MON. The window is passed via the mail system to the user, Jane, who is to have the DEVELOPER capacity.

```
: CREATE_WINDOW CTL_MON_WINDOW,  -- Set up a window for Jane.
     TARGET=>COCKPIT,
     PARTITION=>(MODULE=>CONTROL_MON),
     CAPACITY=>DEVELOPER

: SEND_MAIL  TO_USER=>JANE,      --  and send it off.
             SUBJECT=>"Please set up Control Monitor System",
             MESSAGE_OBJ=>CTL_MON_WINDOW
```

Similarly, a window on COCKPIT with the capacity TESTER is sent to user George. George is given permission to be a TESTER for all modules with a TEST_LEVEL of 1.

```
: CREATE_WINDOW LEV_ONE,         -- Set up a window for George.
     TARGET=>COCKPIT,
     PARTITION=>(TEST_LEVEL=>1),
     CAPACITY=>TESTER

: SEND_MAIL  TO_USER=>GEORGE,    --  and send it off.
             SUBJECT=>"Please checkout the Cockpit System",
             MESSAGE_OBJ=>LEV_ONE
```

... Some time later, the manager checks on George's testing:

```
: LIST_PARTITION COCKPIT.(MODULE=>CONTROL_MON,TEST_LEVEL=>1)
  Partition  COCKPIT.(MODULE=>CONTROL_MON,TEST_LEVEL=>1)
  (SUB_MODULE=>INITIALIZATION,TYPE=>DOC)
  (SUB_MODULE=>INITIALIZATION,TYPE=>SOURCE)
  (SUB_MODULE=>STICK_POS_INPUT,TYPE=>DOC)
  (SUB_MODULE=>STICK_POS_INPUT,TYPE=>SOURCE)
  (SUB_MODULE=>STICK_POS_INPUT,TYPE=>TEST_RESULTS)
```

... The manager checks the test results for the stick position
... sub module:

```
: LIST COCKPIT.CONTROL_MON.STICK_POS_INPUT.TEST_RESULTS.1
  Test 1 Ok
  Test 2 Fail  THETA_WARN=30, THETA_STICK=37
  Test 3 Ok
  Test 4 Fail  THETA_WARN=18, THETA_STICK=-22
  Test 5 Ok

: LOGOUT     -- Seeing that work is under way, the manager signs off.
```

BIBLIOGRAPHY

Government Documents

[S.O.W.] Rome Air Development Center: Revised Statement of Work for Ada Integrated Environment, PR No. B-0-3233, 80MAR26.

[STONEMAN] U.S. Department of Defense: Requirements for Ada Programming Support Environments, "STONEMAN". (J. Buxton and V. Sterning, ed.) February 1980.

[LRM] U.S. Department of Defense: Reference Manual for the Ada Programming Language, (Proposed Standard Document). July 1980 (reprinted November 1980).

Non-Government Documents

[Cashman 80] P.M. Cashman and A. W. Holt: "A Communications-Oriented Approach to Structuring the Software Maintenance Environment". ACM Software Engineering Notes, January 1980.

[Cheatham 79] T. Cheatham, J. Townley, and G. Holloway: "A System for Program Refinement". Proceedings of the 4th International Conference on Software Engineering, 1979.

[Fostel 80] G. N. Fostel: "A Data Structure Methodology for Technology Transfer". Intermetrics, Inc., 1980.

[Goos 81] G. Goos and Wm. A. Wulf: Diana Reference Manual. Report of Institut Fuer Informatik I1 Univeritaet Karlsruhe and Computer Science Department, Carnegie-Mellon University, March 1981.

[Habermann 80] A. N. Habermann and I. R. Nassi: "Efficient Implementation of Ada Tasks". Carnegie-Mellon University technical report, January 1980.

[Horsley 79] T. Horsley and W. Lynch: "Pilot: A Software Engineering Case Study". Proceedings of the 4th International Conference on Software Engineering, 1979.

[Intermetrics 80] Intermetrics, Inc.: HAL/S-360 User's Manual. IR-360-2, May 1980. (Prepared for Johnson Space Center, Houston, Texas).

[Jessop 76] W. H. Jessop, J. R. Kane, S. Roy, and J. M. Scanlon: "ATLAS - An Automated Software Testing System". Proceedings, 2nd International Conference on Software Engineering, October 1976.

[Kernighan 76] B. Kernighan and P. Plauger: Software Tools. Addison-Wesley, 1976.

[Lauer 79] H. Lauer and E. Satterthwaite: "The Impact of Mesa on System Design". Proceedings of the 4th International Conference on Software Engineering, 1979.

[Leverett 80] B. W. Leverett, R.G.G. Cattell, S.O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, W. A. Wulf: "An Overview of the Production-Quality Compiler-Compiler Project". Computer, Vol. 13, No. 8 (August 1980).

[Loveman 77] D. B. Loveman: "Program Improvement by Source-to-Source Transformation". Journal of the ACM, Vol. 24, No. 1 (January 1977).

[McGuffin 79] R. McGuffin, A. Elliston, B. Tranter, and P. Westmacott: "CADES - Software Engineering in Practice". Proceedings of the 4th International Conference on Software Engineering, 1979.

[Millstein 77] R. Millstein: "The National Software Works: A Distributed Processing System". Proceedings of the ACM National Conference, 1977.

[Nestor 81] J. R. Nestor, W. A. Wulf, D. A. Lamb, "IDL - Interface Definition Language User's Manual" (Preliminary Draft), Carnegie-Mellon University, January 1981.

[Notkin 79] D. Notkin and A. N. Habermann: "Software Development Environment Issues as Related to Ada". Proceedings of the Ada Environment Workshop, November 1979.

[Orgass 79] R. J. Orgass: "Converting DEC-10 SIMULA Programs to CMS SIMULA". Department of Computer Science, Virginia Polytechnic Institute and State University technical report, July 1979.

[Sedlik 70] H. Sedlik: Jigs and Fixtures for Limited Production. Society of Manufacturing Engineers, Dearborn, Michigan, 1970.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DA
FIL